Computer Science Education
Vol. 18, No. 2, June 2008, 67–92

Routledge
Taylor & Francis Group

Debugging: a review of the literature from an educational perspective

Renée McCauley[a]*, Sue Fitzgerald[b], Gary Lewandowski[c], Laurie Murphy[d], Beth Simon[e], Lynda Thomas[f] and Carol Zander[g]

[a]*Department of Computer Science, College of Charleston, Charleston, USA;* [b]*Department of Information and Computer Sciences, Metropolitan State University, St. Paul, USA;* [c]*Department of Mathematics and Computer Science, Xavier University, Cincinnati, USA;* [d]*Department of Computer Science and Computer Engineering, Pacific Lutheran University, Tacoma, USA;* [e]*Department of Computer Science and Engineering, University of California, San Diego, USA;* [f]*Department of Computer Science, Aberystwyth University, Aberystwyth, UK;* [g]*Department of Computing and Software Systems, University of Washington, Bothell, USA*

This paper reviews the literature related to the learning and teaching of debugging computer programs. Debugging is an important skill that continues to be both difficult for novice programmers to learn and challenging for computer science educators to teach. These challenges persist despite a wealth of important research on the subject dating back as far as the mid 1970s. Although the tools and languages novices use for writing programs today are notably different from those employed decades earlier, the basic problem-solving and pragmatic skills necessary to debug them effectively are largely similar. Hence, an understanding of the previous work on debugging can offer computer science educators insights into how to improve contemporary learning and teaching of debugging and may suggest directions for future research into this important area. This overview of the debugging literature is organized around four questions relevant to computer science educators and education researchers: What causes bugs to occur? What types of bugs occur? What is the debugging process? How can we improve the learning and teaching of debugging? We conclude with suggestions on using the existing literature both to facilitate pedagogical improvements to debugging education and to offer guidance for future research.

Keywords: debugging; bugs; errors; programming; pedagogy; education

1. Introduction

We began this review of the debugging literature as part of an ongoing computer science education investigation into the problems experienced by novices as they learn to program. In 2001 McCracken and colleagues assessed the programming competency of first year programming students, concluding that many introductory students cannot write correct code (McCracken et al., 2001). A similar study by Lister et al. (2004) suggested that novices also had difficulties reading code. Anecdotally, we have observed that many beginning students experience difficulties while debugging code and so we began to investigate this problem by looking at previous work conducted in this area. We were

*Corresponding author. Email: mccauleyr@cofc.edu

ISSN 0899-3408 print/ISSN 1744-5175 online

DOI: 10.1080/08993400802114581
http://www.informaworld.com

impressed by the depth and breadth of research on debugging, dating as far back as 1973. Reports included investigations of the differences exhibited by novice and expert programmers, exhaustive (and exhausting) categorizations of bugs and causes of bugs, mental models of programming and debugging, and the design and implementation of intelligent tutoring systems. Our aim in writing this review is to summarize the body of debugging literature, primarily for the benefit of other computer science educators and education researchers.

We chose to narrow our focus to topics most relevant to our perspective as computer science educators, in particular, as teachers of programming. We concentrated primarily on novice programmers, although studies comparing novice and expert debugging have been included. We specifically excluded visualization and the design and implementation of intelligent tutoring systems and online debugging systems in general as beyond the scope of this paper. Since debugging is a natural component of programming, much of the work we read was grounded in studies of novice programming. In several instances studies investigating how novices program necessarily incorporated investigations of novice debugging. We mention these more general programming studies only when they add specific insights into debugging.

It is surprising how little page space is devoted to bugs and debugging in most introductory programming textbooks. The Barnes and Kölling (2002) text is unusual in including an entire chapter on this topic in its initial and subsequent editions. In common with them we define *debugging* as an activity that comes after testing where we ''find out exactly where the error is and how to fix it.'' Syntax errors clearly are a problem for novices, but this paper concentrates on the literature concerned with run-time semantic errors (errors that allow compilation but cause abnormal termination at run-time) and logic errors (errors that allow compilation and running but lead to incorrect results).

To organize this review we grouped the research papers according to their relevance in providing insight into one of four focal questions.

- Why do bugs occur?
- What types of bugs occur?
- What is the debugging process?
- How can we improve the learning and teaching of debugging?

We believe investigating these questions facilitates a better understanding of novice debugging, can lead to improvements in debugging instruction, and reveals directions for future research on debugging education.

This review devotes a section to each of these questions. In Section 2, ''Why do bugs occur?,'' we survey the literature that addresses programmers' thought processes that lead to ''buggy'' programs. We are not interested in the simple answers such as ''the programmer creates a defect in the code'' (Zeller, 2006, p. 1) or ''Like natural bugs, they're everywhere'' (Metzger, 2004, p. 1). In both of these books the ''whys'' are not very important since the goal of the authors is to assist the practitioner in removing such bugs. These authors are resigned to the fact that bugs are, indeed, ''everywhere'' and simply need to be fixed. As teachers of novice programmers we believe the question of what novice students are thinking when they produce bugs is much more relevant; students learn from their mistakes only when the causes of the faulty mental models causing the errors are understood.

In Section 3, ''What types of bugs occur?,'' we review the body of research investigating the kinds of errors made, which are often presented as bug categorizations. Researchers have investigated bug types for different reasons: some used their results to

create debugging tools while others have sought to gain insight into the programming process.

In Section 4, ''What is the debugging process?,'' we summarize investigations into the ways in which successful and unsuccessful debuggers approach the problem of removing bugs. Contrasting these two groups has important implications for debugging education, since it suggests the types of thinking and behaviors educators may want to encourage to improve the learning and teaching of debugging for all students. As part of this section we look at the knowledge and strategies that are necessary for successful debugging and we also present the wide body of work where researchers have studied novices and experts to determine how their debugging processes differ.

In Section 5, ''How can we improve the learning and teaching of debugging?,'' we relate what has been discovered through various interventions designed to facilitate debugging instruction.

Clearly, the questions above are as closely related as the papers that seek to answer them. In some cases different facets of the same study are discussed in multiple sections. In the conclusion we present our suggestions for how the existing literature might be used, both to facilitate pedagogical improvements in debugging education and to offer implications for future research.

2. Why do bugs occur?

Why do programmers write buggy programs? The works discussed in this section provide evidence and theories about the mental process of programming that leads to bugs. Most of the results reported came from researchers studying the programming processes and strategies of novices for the purpose of learning how to teach programming more effectively. The majority of these studies took place in the 1980s, the most productive decade of work on debugging to date.

In the early 1980s the Cognition and Programming Group at Yale University, under the direction of Eliot Soloway, studied programmers and programming; specifically, they studied the programming processes used by expert and non-expert programmers. Their work viewed programming as a *goal/plan* process: in its simplest terms, one has a goal and a plan on how to achieve that goal. Programming plans were defined as pieces of code that work together to achieve a desired result or goal. When the plan breaks down a bug occurs, preventing the achievement of the goal (Johnson & Soloway, 1984; Soloway & Ehrlich, 1984; Soloway, Ehrlich, & Bonar, 1982; Spohrer, Soloway, & Pope, 1989). In a study focusing on goal/plan analysis they compared the first compiled versions of programs submitted with later versions and confirmed that ''breakdowns'' between goals and plans led to many of the bugs found (Spohrer, Soloway, & Pope, 1989).

In a similar study, Bonar and Soloway (1985) offered one explanation for how the breakdowns occur. They observed that novice programmers inappropriately applied natural language step-by-step knowledge which led to bugs. Novices brought this ''preprogramming knowledge'' with them to their first programming experiences/courses. For example, the semantics of the word ''while'' in natural language, which is continuously and implicitly updated, differs from the procedural meaning of ''while'' in programming languages, which is explicitly updated when the condition is actually tested at a single point in execution.

Leveraging the goal/plan model, Spohrer and Soloway (1986a) analyzed high frequency bugs, trying to understand what the student was (or was not) thinking at the time the bug was made. Popular belief at the time attributed most bugs to student

misconceptions about language constructs. Spohrer and Soloway identified problems that may negatively influence a novice's understanding of constructions: *data-type inconsistencies* – for example, white space is ignored when integers are input, but is considered part of the data when characters are input; *natural language problems* – see the findings of Bonar and Soloway (1985) mentioned above; *human interpreter problems* – novices expect the computer to be able to interpret a construct in the way they intended it.

However, debunking the notion that misconceptions about language constructs are the cause of most bugs, Spohrer and Soloway found that the majority of bugs are due to other underlying problems. Based on their analyses, they identified seven causes of bugs.

- *Boundary problems* include off-by-one bugs (i.e. loops that terminate one iteration too early or too late).
- *Plan dependency problems* describe misplaced code often related to nesting, such as output statements that should be within a specific clause of an if-then statement rather than after the completed if-then construct.
- *Negation and whole part problems* are related to misuse of logical constructs, such as using an OR when an AND is needed.
- *Expectations and interpretation problems* are misinterpretation of how certain quantities are calculated.
- *Duplicate tail digit problems* involve dropping the final digit from a constant with duplicated tail digits.
- *Related knowledge interference problems* occur when correct knowledge is similar to incorrect knowledge and the two can be confused.
- *Coincidental ordering problems* occur when arithmetic operations are to be carried out in the order in which they appear (left to right) but parentheses are still necessary to override operator precedence.

Their findings suggest that educators can help novices by making them aware of the types of non-construct-based problems they may experience. The actual bugs they found during their study are discussed in Section 3.

Like the Yale group, Perkins and Martin (1986) investigated the thinking of programmers and sought to understand why errors occurred. In their study they observed and interacted with high school students as they programmed in an introductory BASIC programming course. They observed that students' difficulty in programming, including finding and removing bugs, was related to their *fragile knowledge*. Fragile knowledge is described as knowledge that students may know partially, have difficultly harnessing, or simply be unable to recall. This knowledge may be related to the specific constructs of a programming language or to more general problem-solving strategies. Four types of fragile knowledge were observed: *missing knowledge* is knowledge that has not been acquired; *inert knowledge* refers to knowledge that the student has but fails to retrieve when needed; *misplaced knowledge* refers to knowledge that is used in the wrong context; *conglomerated knowledge* is a misuse of knowledge in which a programmer combines two or more known structures incorrectly. Misplaced and conglomerated knowledge are similar and sometimes indistinguishable.

Pea (1986) agreed with Bonar and Soloway that programming errors result from misconceptions held by programmers and focused attention on language-independent misconceptions. Pea argued that bugs arise due to students' overall misconception that writing programming instructions is analogous to conversing with a human. He identified three types of conceptual bugs: parallelism bugs; intentionality bugs; egocentrism bugs.

The *parallelism bug* is based on an incorrect assumption that different lines of code can be known to the computer or executed simultaneously. Pea noted that the natural language ''while'' bug mentioned by Bonar and Soloway is an example of this. The *intentionality bug* occurs when a student incorrectly attributes foresightedness to the program, thinking that it ''goes beyond the information given'' as it executes. An *egocentrism bug* is ''where students assume there is more of their meaning for what they want to accomplish in the program than is actually present in the code they have written.'' Pea suggested ''these classes of conceptual bugs are rooted in a 'superbug,' the default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretive powers.''

Recently, Ko and Myers (2005) offered a more formal framework for understanding errors. Contending that software errors are not wholly a result of cognitive failure on the part of the programmer but are also due to a variety of ''environmental factors'' related to the programming system (such as poorly conceived programming constructs, problems with the programming environment, etc.) and the external environment (such as interruptions), their work presents a framework for describing programming errors which identifies three general programming activities that programmers perform: specification activities (design and requirements); implementation activities (code manipulation); run-time activities (testing and debugging).

The programming system provides different interfaces for different activities. For example, a debugger and output screen are examples of interfaces for run-time activities, while the editor is an example of an interface for implementation activities. Some sort of information is acted upon during each activity (such as requirements specifications during specification activities). Ko and Myers identified six actions that programmers perform when interacting with the various programming system interfaces: design; creation; reuse; modification; understanding; exploration. All of these actions occur in ''programming activities,'' while fewer occur in other activities. They identified three types of breakdowns that result from a programmer's ''cognitive limitations'' in concert with the programming system or external environment: skill breakdowns; rule breakdowns; knowledge breakdowns. These breakdowns can occur during any activity. Furthermore, they defined a cognitive breakdown as consisting of four components spanning cognition and the programming system: the type of breakdown; the action being performed when the breakdown occurs; the interface on which the action is performed; the information that is being acted upon. Software errors result when ''chains of cognitive breakdowns are formed over the course of programming activity.''

Why do bugs occur? Summarizing our cited works, the literature seems to agree that the programmer experiences a breakdown. Some, but not most, of those breakdowns are caused by misconceptions about language constructs. Most errors seem to result from a chain of cognitive breakdowns. These breakdowns occur in skill, rules, or knowledge. Rule breakdowns cause errors in carrying out programming plans. Within rule breakdowns one might either apply the wrong rule or a bad rule; bad rules particularly may result from fragile knowledge. Knowledge breakdowns may result from both fragile knowledge and the superbug causing incorrect programming plans. Thus, while there is no simple answer to the question, the findings cited offer useful insights.

3. What types of bugs occur?

A significant body of the debugging literature is focused on identification of specific bugs and categorizations of such bugs. In 1994 the IEEE published a comprehensive list of bug

classifications to help professional software developers identify and track software anomalies (IEEE, 1994). In some studies researchers have attempted to categorize common bugs and then use the categorizations in the development of a tool to assist novices (Hristova, Misra, Rutter, & Mercuri, 2003; Johnson, Soloway, Cutler, & Draper, 1983). Others categorized bugs to gain a better understanding of the problems that novice programmers encountered (Ahmadzadeh, Elliman, & Higgins, 2005; Robins, Haden, & Garner, 2006; Spohrer, Pope, Lipman, et al., 1985). Finally, some categorizations simply emerged as common knowledge from teaching experience or personal experience in building a large software system. (Ford & Teorey, 2002; Hristova et al., 2003; Knuth, 1989). These categorizations also led to questions about the frequency with which different bugs occurred, as well as which bugs are considered the hardest to find.

These categorizations have been derived through a variety of methods, including surveys, interviews, think aloud exercises, researcher observation of student behavior, expert self-observation, hand analysis of program listings, and examination of compilation records. The numbers of categories have varied greatly, from as few as six when they focused only on compiler-detected errors (Ahmadzadeh et al., 2005) to over 100 derived during a large-scale analysis of novices' code (Johnson et al., 1983; Spohrer, Pope, Lipman, et al., 1985). Categorizations created to serve primarily as debugging aids have had more manageable numbers of categories, around 10–30. This section summarizes these bug categorizations.

The Yale University Cognition and Programming Group was responsible for some of the earliest attempts to categorize students' programming bugs. With the goal of developing PROUST, an automatic program understanding and error diagnosis system, Johnson et al. (1983) created a catalog of novice programming bugs that were non-syntactic in nature. Their *Bug catalogue: I* describes the bugs found through hand inspecting 261 introductory Pascal students' first syntactically correct solutions to a typical programming problem. Using the goal/plan model for programming, their analysis was based on comparing correct programming plans representing "stereotypic action sequences in programs" with actual code. The catalog summarizes the numbers and types of bugs which deviate from a correct plan, classifying each as being one of missing (not present but should be), spurious (present but should not be), misplaced (present but in the wrong place), or malformed (present but incorrectly implemented). Within the classification the 783 bugs detected were divided into eight categories depending upon the type of statement in which they occurred: inputs; outputs; initializations (assignment statements); updates (assignment statements); guards; syntactic connectives (block delimiters); complex plans; declarations.

The original bug catalog was followed by *Bug catalogue: II, III, IV* (Spohrer, Pope, Lipman, et al., 1985), a similar document describing the bugs in solutions to three typical programming assignments from an introductory programming course at Yale in spring 1985. More important than the actual enumeration of these catalogs may be the other findings that resulted from this work. Among the most notable is Spohrer and Soloway's (1986a) finding that "a few types of bugs account for a majority of the mistakes in students' programs." They analyzed 61 students' first compiled versions of solutions for three program assignments in an introductory Pascal programming course. They discovered that 10% (11 out of 101) of the bug types accounted for between 32% and 46% of the bug occurrences in each program. Of these 11 most common bugs (8 different types) only 1 was clearly due to a misunderstanding of a language construct. Specifically, some students misunderstood how the readln statement worked on input of character data, assuming blanks would be ignored as they are with numeric data. Three other bugs

(two bug types) were identified as possibly being due to construct misunderstandings: two occurrences of an *or-for-and* error, where an OR logical connective was used when an AND was needed; one occurrence of a *needs parentheses* error, where parentheses were needed for a formula to be computed properly. For both of these bug types the authors offer plausible, non-construct-related reasons why they may have occurred.

The seven bugs identified as being non-construct-based fell into five different types, as can be seen in Table 1. (See Section 2 for descriptions of the plausible causes of bugs.)

About 20 years later Ahmadzadeh et al. (2005) categorized compiler-detected programming errors in a two part study of novice debugging. In their first experiment they examined 108,652 records of student errors produced by 192 students in an introductory programming course. They only looked at errors that could be found by a compiler, but they also observed that a few (6) of the many (226) distinct semantic errors detected comprised 50% of all semantic errors found. The six common semantic errors detected were: variable not declared; inappropriate use of a non-static variable; type mismatch; non-initialized variable; method call with wrong arguments; method name not found. Though not fitting our concentration on run-time semantic and logic errors, these semantic errors must be faced by students before any other debugging begins.

Some bug categorizations have been derived through observational or anecdotal means. In *Practical debugging in C++* Ford and Teorey (2002) identified the 32 most common bugs in first programs. This list is based on their observations during a decade of teaching introductory programming. About half of the bugs listed were syntax related. The most common non-syntax-related errors identified were: infinite loops; misunderstanding of operator precedence; dangling else; off-by-one error; code inside a loop that does not belong there; not using a compound statement when one is required; array index out of bounds.

In 2003 Hristova and colleagues used surveys of faculty, teaching assistants, and students, as well as their own experience, to identify common Java errors and misconceptions (Hristova et al., 2003). Their list included 13 common syntax errors (e.g. $==$ versus $=$ and mismatched parentheses), one semantic error (invoking class methods on objects), and the following six logic errors: improper casting; invoking a void method when you want a result; failure to return a value; confusion regarding declaring

Table 1. Spohrer et al.'s (1985) types of non-construct-based bugs.

Type of bug	Description	Cause
Zero is excluded	Zero not considered a valid amount	Boundary problem
Output fragment	Output statements were improperly placed in the code, executing when they should not have: missing implication of if/else placing code outside the begin/end block	Plan-dependency problem
Off-by-one	Use of an incorrect guard bound (for example 60, rather than 59) Incorrect relational operator ($<=$ rather than $<$)	Boundary problems
Wrong constant	Incorrect constant typed (several programmers used 4320 when they should have used 43200)	Duplicate tail-digit problem
Wrong formula	The formula used to compute a value was incorrect	Expectations and interpretation problem

parameters in methods and providing them in the call; a class declared abstract due to omission of function.

In 1989 Donald Knuth, legendary for his attention to detail, used his handwritten code and a bug-by-bug log to classify every change he made to TeX over 10 years into one of 15 descriptive categories (Knuth, 1989). Nine of these categories describe situations we consider bugs: algorithm awry – your original solution is insufficient or incorrect; *blunder or botch* – a mental typo; you know what you wanted to write, but you wrote something else; *data structure debacle* – your use of variables and structures is incorrect or insufficient for the problem; *forgotten functionality* – you forgot to put in a crucial bit of functionality, such as incrementing a counter in the loop; *language liability* – a misunderstanding of a language construct; *mismatch of modules* – problems such as parameters out of order or type mismatches in the calls; *robustness* – not handling erroneous input; *surprise scenario* – bugs resulting from unforeseen interactions between parts of the program; *trivial typos* – stupid mistakes of typing (e.g. – for +) not caught by the compiler. Knuth's work is interesting because it is a self-study, revealing the perspective of an experienced programmer rather than a novice. His categorizations reflect the discussion of Section 2: the categories have a sense of the goal–plan model used by Soloway's group (e.g. algorithm awry/malformed, forgotten functionality/missing), but also reflect the notion that bugs may occur because of skill breakdowns (blunders, trivial typos) or language conception difficulties, as discussed in Ko and Myer's framework.

More recent work includes that of Robins et al. (2006). In a study of novices on an introductory programming course data on the kinds of problems students encountered while working in laboratory sessions were collected in 2003 and 2004. The students were enrolled on a course that teaches Java, which consisted of 26 fifty minute lectures and 25 two hour laboratory sessions. This research is of particular interest as it studied students learning object-based programming in Java. The problems encountered were categorized as one of three general types: background problems; general problems; language-specific problems. *Background problems* are related to the use of tools, understanding the task, and very basic design issues, such as knowing how to get started or organize the solution to the problem. *General problems* include difficulties with basic program structure, object concepts, naming things, and trivial mechanics, like mismatched parentheses and typos. *Language-specific problems* include issues related to control flow, loops, selection, Booleans and conditions, exceptions, method signatures and overloading, data flow and method header mechanics, IO, strings, arrays, variables, visibility and scope, expressions and calculations, data types and casting, reference types, class versus instance, accessors/modifiers, constructors, hierarchies, GUI mechanics, and event-driven programming. In both years problems related to trivial mechanics dominated. It is also notable that problems related to understanding the task and basic design occurred more often than those specifically related to language.

Hanks (2007) replicated the work of Robins et al. but studied novices working in pairs, rather than alone. McDowell, Werner, Bullock, and Fernald (2006) showed that students working in pairs are more successful, more confident, and more likely to continue studying computing. Hanks found that the problems encountered by students working in pairs were essentially the same in type as those encountered by students working alone. However, he found that students working in pairs required less instructor assistance in working through the problems than did students working alone.

In addition to knowing which bugs occur frequently, it is interesting to note which of the bugs have been found hardest to detect in code. Gould (1975), in a study of expert programmers, found that bugs in assignment statements were much more difficult to find

than those in loop conditions or array accesses. He suggested that detecting an error in an assignment statement requires a programmer/debugger to understand the purpose of the entire program, while array and loop condition bugs can often be solved with information local to the use of that particular construct (such as by examining local boundaries). Gould's work is addressed in the next section "What is the debugging process?"

What types of bugs occur? Through the lens of goal–plan analysis we see that most bugs are the result of something missing or malformed. Using Knuth's categories, which provide the point of view of an experienced programmer, most of the novice bugs reported here are either language liabilities or algorithm awry. In the case of the Ahmadzadeh et al. and Hristova et al. studies most of the errors are language liabilities, not surprising since the first studied compiler errors and the second set out to specifically collect Java errors. Some bugs appear to be endemic to novice programming, regardless of language or paradigm [e.g. off-by-one errors, operator precedence errors, misplacement of code in (or out of) a loop] and were noted by both Spohrer et al. and by Ford and Teorey. However, little research has been done on language/compiler-independent errors in object-oriented programming. Along with developments in the use of new languages and environments (e.g. Python, Alice, Scratch, and Ruby), this suggests new investigations into bugs currently plaguing novices may be in order.

4. What is the debugging process?

Programming educators have a great deal of expertise in debugging student programs, gained through years of experience and grounded in substantial content knowledge that goes well beyond the scope expected in introductory programming students. Thus, this expertise may not be an effective guide to teaching novices to debug.

Studying other successful debuggers, particularly novices, may provide useful insights for teaching, because the ways those debuggers understand, diagnose, locate, and correct bugs can offer more accessible models of how beginners could be taught. Furthermore, from a constructivist perspective (Ben-Ari, 1998) educators should not ignore the incorrect mental models that are the root causes of unsuccessful debugging, but must understand and correct their causes in addition to teaching correct debugging techniques. This section reviews investigations of the knowledge and strategies of both successful and unsuccessful debuggers.

We found that research investigating the debugging process generally focused on two main components: the types of knowledge or reasoning necessary for successful debugging; the specific strategies that debuggers, human or automatic, employed when they debugged. Consequently, we refine our discussion of work in this section using the following questions: "What types of knowledge are aids in debugging?", "What strategies are employed in debugging?", and "How do novices and experts differ?"

4.1. What types of knowledge are aids in debugging?

In this subsection we focus on aids to the debugging process. These range from concrete artifacts, such as program output (Gould, 1975; Gould & Drongowski, 1974), to more general types of knowledge, such as an understanding of the problem domain (Ahmadzadeh et al. 2005; Ducassé & Emde, 1988).

An early investigation of debugging strategies by Gould and Drongowski (1974) and a similar follow-up study by Gould (1975) laid much of the groundwork for the debugging research that followed. They observed 10 expert programmers as they attempted to debug

several programs, each of which compiled but contained a single bug that caused it to run incorrectly. The protocol used multiple occurrences of several FORTRAN programs, with a different bug each time. Subjects were provided with a code listing and formatted, labeled I/O printouts and told to mark the listings to indicate the order in which they studied the code or printouts and to record any notes or thoughts. They were also given access to an interactive debugging system with the source code and input data files. The study protocol required that subjects inform the researcher whenever they located a bug, allowing researchers to track the time needed to find the bug. Researchers collected code listings with the notes made by the subjects and the subjects were interviewed immediately after the debug session. From this data Gould and Drongowski learned what types of information or knowledge their expert subjects relied upon during debugging.

Most subjects used the output data in developing a hypothesis about a bug; few reported using the input data. The usefulness of output in the debugging process was also reported in studies by Katz and Anderson (1987) and Carver and Risinger (1987). The experts also found the presence of meaningful variable names to be useful; they made assumptions about meanings of variables based on those names, although they did not always verify that their assumptions were correct. Program comments were used and application domain knowledge made it easier to detect bugs. Gould observed that the time spent debugging a program decreased the second time the program was seen. This result, which suggests the importance of program comprehension, was confirmed in later studies by Katz and Anderson (1987), Ahmadzadeh et al. (2005), Gugerty and Olson (1986), and Nanja and Cook (1987).

In 1988 Ducassé and Emde used Gould's work as a framework to review 18 automated debugging systems and 12 cognitive studies. They identified seven types of knowledge used during the bug location phase of debugging:

- knowledge of the intended program (program I/O, behavior, implementation);
- knowledge of the actual program (program I/O, behavior, implementation);
- an understanding of the implementation language;
- general programming expertise;
- knowledge of the application domain;
- knowledge of bugs;
- knowledge of debugging methods. (Ducassé & Emde, 1988)

They observed that not all types of knowledge are required for every debugging session and that the wide range of knowledge needed makes it difficult to incorporate it all into a single debugging tool.

Ahmadzadeh, Elliman, and Higgins conducted two experiments with novice programmers in which they identified some subjects as "good programmers" or "weak programmers" and also as "good debuggers" or "weak debuggers" (Ahmadzadeh et al., 2005). Among the novices studied they found that a majority of good debuggers were also good programmers, but not all good programmers were good debuggers. This contradicted the finding by Grant and Sackman (1987) that programmers good at one programming activity were usually good at others. Ahmadzadeh et al. also found that the good debuggers possessed the seven types of knowledge enumerated by Ducassé and Emde. Studying the good programmers who were identified as weak debuggers they discovered that the weakness was partly due to a lack of knowledge of the program implementation as they were debugging code written by others, not their own. This supported Gould's (1975) finding related to code familiarity (i.e. program comprehension).

The weak debuggers also suffered from an inability to localize bugs because they lacked knowledge of debugging strategies.

4.2. What strategies are employed in debugging?

A considerable amount of research has been aimed at explicating programmers' debugging strategies to gain a deeper understanding of the debugging process.

Gould and Drongowski (1974) and Gould (1975) observed that experts followed the following steps when debugging. First, a debugging tactic was chosen (e.g. some started with the output listings, some started at the first line of code in the listing and read it line by line until something suspicious was detected). The use of a tactic, whatever it was, led to a clue (something suspicious in one of the information sources), to a bug, or to neither. If a clue was strong enough, the subject may have reported that the line containing the clue contains the error. If a clue was not strong enough, it likely led to the formulation of some hypothesis that led the subject to select another debugging tactic. If nothing suspicious was detected using a particular debugging tactic (i.e. no clue was found), this may have provided the subject with useful information that led them to generate a hypothesis or to choose another debugging tactic and start the cycle again.

Vessey asked 16 practicing programmers to debug a COBOL program, given a physical copy of the program listing, input data, correct output data, and the incorrect output data produced by the program (Vessey, 1985). A verbal protocol procedure was used to identify debugging *episodes* – a group of phrases or assertions about tasks related to the same programming goal or objective. From these observations Vessey identified a hierarchy of goals subjects used:

- determine the problem with the program (compare correct and incorrect output);
- gain familiarity with the function and structure of the program;
- explore program execution and/or control;
- evaluate the program, leading to a statement of hypothesis of the error;
- repair the error.

Vessey observed that subjects followed a complete "strategy path" while debugging. A complete strategy path consisted of a choice between two options from each of four distinct intermediate strategy paths, resulting in 16 possible complete strategy paths, seen in Figure 1.

Katz and Anderson (1987), in four interrelated studies with 18–40 LISP students, studied the debugging process as a particular case of troubleshooting. Troubleshooting identified four stages: understand the system; test the system; locate the error; repair the error. Their studies provided strong evidence that for debugging these stages are separate, and in particular the skills needed to understand the system are not necessarily connected to the skills needed to locate the error.

In the "understand the system" stage Katz and Anderson provided evidence that students take longer to understand a system when the system is not one that they coded. This agreed with the results in Gould (1975), described in section 4.1. In the "locate the error" stage they differentiated between three different techniques for finding an error:

- forward reasoning program order (related to simulating the program's execution);
- forward reasoning serial order (the order the lines appear as the code is read);
- backward/causal reasoning (working back from the output).

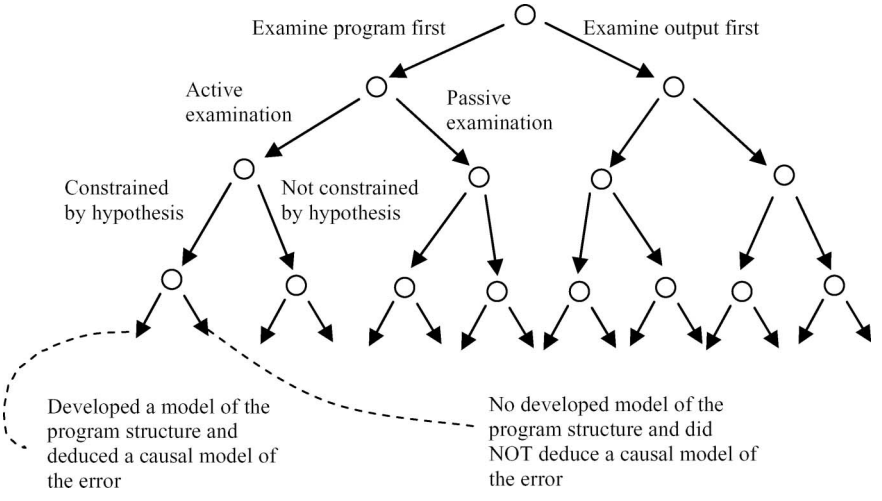

Figure 1. Vessey's possible debugging strategy paths.

Katz and Anderson found that when trained in a particular technique students tended to use it, that causal reasoning, although slower, led to more accurate results overall, and that people used different techniques when debugging their own programs than when debugging programs written by someone else. Debuggers were more likely to use forward reasoning when searching another's code and backward reasoning if working with their own. Katz and Anderson speculated that this difference was due to mental models built by students while writing their own programs. Moreover, they found that the skills required in the "repair the error" stage were unrelated to methods utilized to locate the error. Their study also established that errors made by more experienced programmers were typically *slips* – bugs not generally repeated and easily fixed when found. This was consistent with Knuth's (1989) report on his TeX errors and with Perkins and Martin's (1986) notion of fragile knowledge. This suggests that for most students with some expertise the difficulty of debugging is not in repairing the error but rather in the earlier stages of the troubleshooting process (understanding the system, testing the system, or locating the error).

Although revealing, some caveats must be kept in mind about the Katz and Anderson study. Their experimental mechanism prevented students from using some error locating techniques, in particular the techniques of printing at every line and commenting out lines. Katz and Anderson emphasized that this did not invalidate their results because they were only interested in the hypothesis that programmers used different strategies and removing choices leads to more homogeneous strategies, since fewer were available. However, this may not be so if the two missing strategies were in fact the most preferred. Also, the subjects were restricted to only fixing the bug – adding bugs was prevented by the experimental system. These somewhat artificial conditions suggest that investigation in a more real-world situation may be useful.

Ducassé and Emde's 1988 study identified four global strategies that are employed, either alone or in combination, in debugging:

- filtering (tracing algorithms, tracing scenarios, path rules, slicing/dicing);
- checking computational equivalence of the intended program and the one actually written (algorithm recognition, program transformation, assertions);

- checking the well-formedness of the actual program (language consistency checks, plan recognition);
- recognizing stereotyped errors, those that can be easily recognized from experience (bug cliché recognition).

Tubaishat (2001) described the conceptual model underlying the architecture of the Bug-Doctor tool designed to aid programmers in finding errors. Based on cognitive science studies, the conceptual model for software fault localization featured both shallow and deep reasoning. Shallow reasoning used test cases, output, a mental model of what a correct implementation would be, and diagnostic rules of thumb. Deep reasoning, which was the focus of Tubaishat's paper, is characterized by intensive use of program recognition. It is supported by a knowledge base called a programming concepts library – a hierarchy of problem domain, algorithmic, semantic, and syntactic programming language knowledge/rules.

A two level approach to code recognition and fault localization occurred in parallel during deep reasoning: coarse-grained analysis and fine-grained analysis. During *coarse-grained analysis* a chunking process dominated both the program recognition and fault localization processes. The programmer recognized individual chunks in the buggy code, analyzed the overall control structure for correctness, and identified the focal points for fine-grained analysis. *Fine-grained analysis* involved more detailed recognition and localization as the programmer examined the code on the micro level.

Carver and Risinger (1987) tested whether or not debugging can be explicitly taught. Their study employed the strategy shown in Figure 2 for finding and fixing bugs when testing a program and finding it not correct.

This strategy is a causal reasoning strategy; it is based on observation of the output, whenever possible. However, even when it backs up into brute force forward reasoning, the thinking in the second stage should lead students into program order reasoning (examining the code in execution order) rather than serial order reasoning (examining the lines as they appear physically in the code).

Although researchers have approached debugging strategies in a variety of ways, we can see connections between the studies. The goals described by Vessey have resonance with Ducassé and Emde's global strategies: determining the problem with the program corresponds to checking computational equivalence and checking well-formedness; gaining familiarity with the function and structure of the program and exploring program execution correspond to filtering; repairing the error corresponds to recognizing stereotyped errors and checking the well-formedness of the actual program. Because Ducassé and Emde concentrated on the bug location aspect of debugging, their strategies were entrenched in the error location stage of the troubleshooting framework suggested by Katz and Anderson. Tubaishat's work connected to Katz and Anderson's understanding the system stage of troubleshooting in the coarse-grained analysis, and to bug location in the fine-grained analysis. The model demonstrated by Carver and Risinger provided strategy choices similar to Vessey's but more closely resembled the overall troubleshooting framework of Katz and Anderson.

4.3. How do novices and experts differ?

Studies of novice and expert differences can be found in the educational literature for nearly any discipline (see, for example, Bransford, Brown, and Cocking, 2000), and debugging is no exception. While it is natural to assume that results found for experts

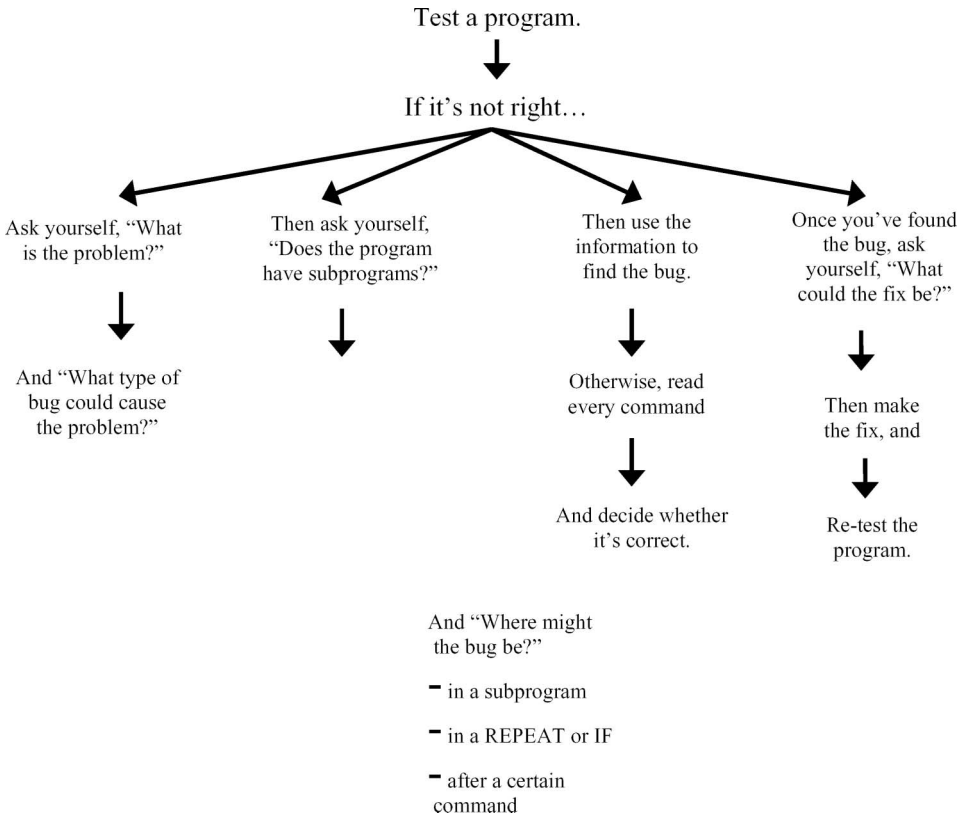

Figure 2. Carver and Risinger's strategy for finding and fixing bugs.

suggest implications for novices, they should be used with caution; expertise is highly complex, develops over time with experience, and is highly contextualized within a rich body of content knowledge.

Several studies have explicitly probed differences in novice and expert debugging processes and behaviors. Four complementary works focusing on differences in the abilities of these two groups, using similar experimental methodologies and reporting comparable results, are those of Jeffries (1982), Vessey (1985), Gugerty and Olsen (1986), and Nanja and Cook (1987). A different approach was taken by Fix, Wiedenbeck, and Sholz (1993), who looked at the abstract representations, or mental models, typically held by experts but not usually by novices.

The first four studies (Jeffries, 1982; Gugerty & Olsen, 1986; Nanja & Cook, 1987; Vessey, 1985) explored similarities and differences exhibited by novices and experts during computer-based debugging sessions. Jeffries identified novices as students at the end of a first programming course. Here experts were advanced graduate students in computer science. All subjects were given unlimited time to find errors in two simple Pascal programs. Experts spent much of their time on comprehension of the program and were able to build more complex mental representations of the programs. As such, experts were able to recall more details of the programs than novices could. Experts were also able to find more bugs and find them faster than the novices did. Interestingly, Jeffries observed

that novices and experts used similar strategies while debugging, but the comprehension skills of the experts were superior. Her study was very small, involving six experts and four novices.

A key component of Vessey's work (see the previous section for the protocol) is that it reported on a set of subjects who were all practicing programmers. Vessey classified programmers as novices or experts according to their ability to ''chunk'' the program they debugged. Vessey found expertise, based on ''chunking ability,'' to be an accurate indicator of debugging strategy and speed. She attributed the more erratic debugging practices of novices to their inability to chunk programs. She also found that experts used a breadth-first approach when debugging. They familiarized themselves with the program, gaining a systems view, before attempting to find the source of an error. On the other hand, novices take a depth-first approach, focusing on finding and fixing the error without regard to the overall program.

Gugerty and Olsen (1986) employed a more traditional distinction between novices and experts, comparing students just finishing a first or second course in Pascal with advanced computer science graduate students. They reported on two complementary experiments which suggested that the superior debugging performance of experts was attributable to their greater program comprehension ability.

In the first experiment all subjects participated in a training session, including a coding experience. Afterwards they were asked to debug three LOGO programs. Each program was syntactically correct and contained only one bug. The results showed that most subjects could find and fix the bug within the allotted time, but that experts found the bug much faster (see Table 2). Using hypothesis testing (operationalized to be a single edit then run process), novices hypothesized much more frequently than experts and were much less likely to test a correct hypothesis on the first try. Gugerty and Olsen also identified executing the code as a viable debugging strategy and showed that experts did this much more frequently than novices. Another finding was that novices were much more likely to introduce new bugs during debugging. Specifically, the novices who never managed to find and fix a program bug 92% of the time introduced at least one new bug.

In contrast to their results with LOGO programmers, a second experiment by Gugerty and Olsen using Pascal found that novices were notably less successful at finding and fixing a program containing a single bug within the allotted time. Those novices who did successfully debug the program were significantly slower than the experts, spending twice as long before making a first hypothesis (24.3 versus 12.9 minutes). Interestingly, novices and experts both devoted the same proportion of time prior to making a hypothesis on the same activities: studying the program and studying a description of the purpose of the program.

Nanja and Cook (1987) investigated expert and novice differences in a study with six each of novice (finishing second term), intermediate (junior level), and expert (graduate) student programmers from a single university. All 18 subjects were asked to debug a

Table 2. Gugerty and Olson's (1986) LOGO programming debugging study results.

	Time to fix bug	Hypothesis testing single edit, then run	Correct hypothesis on first try	Executing code
Novices	18.2 min	3.6 per program	21%	once every 2.9 min
Experts	7.0 min	1.6 per program	56%	once every 1.9 min

Pascal program that read data, bubble sorted it, and then performed several binary searches. The program was seeded with six common novice errors: three of these were run-time semantic and three were logic. The study's main conclusion, similar to those of Gugerty and Olsen, suggested experts were much better at debugging by comprehension while novices worked in isolation to track down bugs one at a time. The authors speculated that this failure of comprehension was due to a tendency of novices to read the code in the order it appeared rather than in execution order, as the experts did. However, unlike Gugerty and Olsen's study, the experts spent longer on their initial reading of the code and half of the novices spent no time on initial reading at all (see Table 3).

All the experts in Nanja and Cook's study successfully debugged the program, while only two of six novices were successful and took much more time to debug. Furthermore, the novices introduced new errors much more frequently than the experts and unlike the experts they did not immediately correct their mistakes. Novices added many more debugging statements than thje experts and, again unlike the experts, did not use online debugging tools. The experts corrected groups of errors before testing their corrections while the novices tended to correct and test one error at a time. The novices also tended to find and fix semantic errors before logic errors. Changes to only six lines of code were required to correct all of the errors. The experts modified an average of 8.83 lines while the novices made three times as many modifications, and frequently nearly completely rewrote functions.

Fix et al. (1993) conducted a study of the mental models (abstract representations of programs) used by 20 experts and novices. They used 11 questions to investigate the differences between expert and novice comprehension of programs. While the testing environment was somewhat artificial – the subjects studied the code, put it away, and then answered the questions – the questions were well-designed to capture differences by providing examples where the expert's superior representation resulted in a more accurate response than that of the novice and other examples requiring a much simpler mental representation in which the experts and novices had comparable responses. The authors found that the experts had a deeper, more sophisticated mental model than the novices. Specifically, the experts:

- have representations that are hierarchically structured and are able to map between layers;
- recognize recurring patterns;
- make connections between entities in the program;
- have a representation that physically relates the code to its purpose.

The differences noted above were confirmed by Bednarik and Tukiainen (2005), whose findings highlighted differences between novices and experts in their use of coarse- and fine-grained analysis. In their experiment with novice and expert subjects programs were

Table 3. Nanja and Cook's (1987) Pascal programming debugging study results.

	Time to debug	Initial reading of code	New errors introduced	Debugging statements added
Novices	56.00 min	1.50 min	4.83	2.83
Experts	19.83 min	4.83 min	1.00	0.83

presented on a screen with most of the code blurred. The user could move a small peephole, through which code could be clearly read, around the code to make fine-grained inspections of the code. Compared with their usual debugging strategies novices were unaffected by the blurring of the code, whereas experts were affected.

The results reported on in this subsection are in many ways unsurprising: experts were typically faster and more effective debuggers than novices. The consensus among researchers is that this is due to the superiority of experts' command of program comprehension strategies. Novices were more likely to work on isolated pieces of code and to create new bugs as they attempted to fix existing bugs. This inevitably leads to the question that motivates the research presented in the next section.

5. How can we improve the learning and teaching of debugging

Can debugging be explicitly taught or is it something that must be learned primarily through experience? In their research into debugging LISP programs Kessler and Anderson (1986) found that "debugging is a skill that does not immediately follow from the ability to write code. Rather ... it must be taught" (p. 208). However, introductory programming texts typically do not adequately address debugging. Moreover, reports on interventions designed to improve students' debugging skills have not been common in recent literature. A few studies suggest that explicit debugging instruction is effective (Carver & Risinger, 1987; Chmiel & Loui, 2004). Other reports have focused on helping students discover their syntax errors using a variety of techniques (Brusilovsky, 1993; Robertson et al., 2004; Tubaishat, 2001; Wilson, 1987).

Carver and Risinger (1987) presented an effective method for debugging instruction. Building on a model for automatically finding bugs in LISP programs, they demonstrated that a curriculum could be designed and taught which resulted in significant improvements in children's debugging abilities. The curriculum incorporated a detailed model of the debugging process via a flow chart that specifically concentrated on bug location and bug repair strategies. Although narrowing the search for bugs was the explicit focus, program comprehension, essential to the application of program order forward reasoning, was also improved.

Carver and Risinger evaluated the impact of their technique by providing 30 minutes of explicit debugging instruction to 18 of 25 Grade 6 students, chosen at random, following 6–8 hours of programming instruction for the entire class. The debugging strategy they taught led students through a hierarchy of questions designed to facilitate bug location and correction and is presented in Figure 2. The 30 minutes debugging instruction resulted in a significant improvement in debugging skills compared with the control group, including skills on a transfer debugging task in which students debugged a real world set of instructions.

A study by Chmiel and Loui (2004) echoed Carver and Risinger's call to explicitly teach debugging. They conducted a study to demonstrate that formal training in debugging helps students develop skills in diagnosing and removing defects from programs. To accomplish this goal in an assembly language course they designed multiple activities to enhance students' debugging skills. These activities included debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments. Students who completed the optional debugging exercises ($n = 27$) reported significantly less time spent on debugging their programs than those who did not ($n = 89$), despite the fact that their overall test scores were not significantly better. Chmiel and Loui were also able to develop a descriptive model of debugging abilities and habits based on

the students' comments in their logs and surveys. They suggested that students and educators could use this model to diagnose students' current debugging skills and take actions to enhance their skills.

Brusilovsky (1993) used human intervention in a closed laboratory situation with 15–16 year olds ($n = 30$) programming in LOGO. Coded exercises were automatically tested, and on each test failure a bell rang, calling an assistant to the student. Debugging assistance was provided as needed in the escalating order:

- demonstrate an instance in which the code fails;
- provide a visual execution of the code;
- explain the code while providing a visual execution;
- provide additional help.

The assistant followed the protocol, trying each of the four assistance types in order, checking after each to see if the student now understood the problem. In this study 84% of the bugs were fixed within the first three steps of assistance. Brusilovsky claimed that this is evidence that program visualization is the key to novice debugging. This study did not report, however, on subsequent changes in student behavior.

In terms of modifying behavior, Wilson (1987) gave anecdotal evidence that a Socratic technique could help novice programmers become more self-reliant in analyzing errors and debugging. The technique requires the instructor to ask a series of ''goal-oriented'' and ''procedure-oriented'' questions, such as ''How do you think that can be done?'', ''How does it work?'', and ''How will it get the results you want?,'' in one-on-one sessions with students who seek help with their programs.

The previously discussed techniques were designed to encourage students to reflect on what was wrong with their programs, rather than immediately rushing in to fix them. In the same vein but turning to tools, Robertson et al. (2004) investigated the use of electronic tools for debugging and encouraging debugging skills by comparing two different interruption styles on end user debugging (these 38 end-users were attempting to debug spreadsheets). The interruption styles compared were: *immediate style* – techniques that interrupt the user and force them to react to the interruption immediately (e.g. a pop-up window that reports an error and requires pressing the OK button before a user can continue); *negotiated style* – techniques that make information available to the user, without forcing them to stop and deal with the information immediately (e.g. the red underline on misspelled words in word processing documents).

Students in the study who were offered negotiated style help were more effective in terms of comprehension and productivity. The authors speculated that the immediate style group were subject to frequent losses of short-term memory, due to the interruptions, and as a result avoided debugging strategies that had high short-term memory requirements. They conjectured that immediate style interruptions would promote an over-reliance on local, shallow problem-solving strategies. While this study used automated intervention, the conclusions are of interest in terms of teaching techniques. Helping students but leaving them to work things out in their own time would appear to be more effective.

This literature review excludes extended discussion of visualization, the design and implementation of intelligent tutoring systems, and online debugging systems in general. We note, however, that our experiences with debugging tools is that many of them use execution traces as a method of assisting students to understand the execution of a program. Tubaishat (2001) characterized the use of execution traces as an example of a shallow reasoning technique. He described, in contrast, the Bug-Doctor system, which

encourages the use of two parallel cognitive processes that occur during deep reasoning. Wilcox, Atwood, Burnett, Cadiz, and Cook (1997) demonstrated that continuous visual feedback is not a panacea and that its utility depends upon the particular programming problem, the type of user, and the type of bug. Clearly, visualization is not directly useful if the novice does not examine the program in a way that makes the bug manifest. This was confirmed by Thomas, Ratcliffe, and Thomasson (2004), who found students often do not use object diagrams in situations where experts would find them useful for visualization.

6. Implications for learning and teaching

The implications of the literature for teaching and learning debugging begin the day students arrive in our classes. The root causes of bugs stem from preconceptions, misconceptions, and the fragility of knowledge and understanding of programming and the language in which they work. The bugs themselves are varied and finding them quickly depends on an ability to exhibit program comprehension on a large scale rather than small chunks. There is evidence that experts do bring particular skills to debugging and that good debugging practices can be taught.

6.1. Combating preconceptions, misconceptions and fragile knowledge

Students' misconceptions or inconsistencies, based on faulty mental models, often lead to unsuccessful debugging (Ben-Ari, 1998). Two particular examples in this review are the pre-programming knowledge discussed by Bonar and Soloway (1985) and the superbug discussed by Pea (1986). Misconceptions are best dealt with as directly as possible. Knowing, for example, that students may misinterpret "while" loops to be continuously, implicitly updated, it is important to present an example demonstrating the difference between this understanding and the actual model in which the condition is explicitly updated and tested at a single execution point.

Similarly, it is important to help students form an abstract model of the computer or computer program, also known as a "notional machine" (Robins, Rountree, & Rountree 2003). Exercises tracing code may help students build this model. Having students predict the output then run the code will allow them to work on self-correcting their own misconceptions. Tracing exercises should be chosen to fight the superbug by emphasizing that program execution is sequential, the order of the statements matters, and statements do not execute in parallel – that the computer does exactly what it is told and does not go as far as what is meant.

Perkins and Martin (1986) suggested approaches that may help students cope with fragile knowledge, including highlighting the functional role of commands by explaining exactly what commands do rather than giving general descriptions. Although it was not in vogue at the time of their experiment, pair programming would seem to naturally incorporate strategies for overcoming fragile knowledge: one student may recall knowledge the other is missing; the pair dynamic will naturally incorporate prompts; partners will require more precise explanations of program execution where an individual student might settle for a more general and less accurate sense of what is happening. Hanks' (2007) study, in which he found that pair programmers ask for help less often than solo programmers, provides some support for this notion.

Contrary to contemporary teaching practice, Wiedenbeck (1985) suggested we encourage students to practice continuously, to the point of "overlearning" basic concepts such as program syntax. Being able to automatically recall the basics is an important

characteristic of experts which enables them to ignore details and engage in higher level problem-solving. Automatic recognition of correct program syntax enables students to focus more energy on constructing correct mental models.

6.2. Building program comprehension skills

Gould and Drongowski (1974), Gould (1975), Gugerty and Olson (1986), Katz and Anderson (1987), Nanja and Cook, (1987), Ducassé and Emde (1988), and Ahmadzadeh et al. (2005) all presented findings revealing the importance of program comprehension in debugging. A study by Lister et al. (2004) indicated this is a skill novices lack. These works suggest a variety of teaching implications.

A commonsense implication is to continue an emphasis on meaningful variable names and comments to help with program comprehension, as discussed in Gould (1975). Ducassé and Emde (1988) noted that experts distinguish between the intended program and the actual program. This suggests designing exercises to encourage students to build program plans that capture the key notions describing the intent of the programs they are writing or reading. Asking questions to force reflection about the overall actions of the code, similar to those used by Fix et al. (1993) to distinguish experts and novices, may help students develop their overall comprehension skills. Nanja and Cook (1987) found that novices read code in the order it was written rather than execution order; exercises that encourage the practice of comprehending a program in execution order will help overall program comprehension. The groundwork for debugging skills may also be laid by providing students with output and asking them to reason about the program based on this output. Vessey (1985) noted the superior chunking ability of experts; activities to encourage chunking would also help increase comprehension skills.

6.3. Explicitly teaching debugging skills

The literature suggests debugging strategies can be taught. Carver and Risinger (1987) effectively used a debugging flow chart to teach children how to debug. Chmiel and Loui (2004) showed that formal training in debugging that includes debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments decreased the time spent on debugging. This suggests explicit instruction in debugging should be fundamental to any beginning programming class. Introductory textbooks would do well to incorporate these ideas and students should be presented with opportunities to practice these skills. Along with a flow chart to provide an overall strategy and debug logs to help students gain familiarity with common bugs, the literature suggests other actions instructors can take.

The skill of forming hypotheses based on erroneous output (Gould & Drogonowski, 1974) and general reasoning about the program based on output appears crucial. Experts' reliance on program output also suggests the value of giving students correct example output with their assignments, or even executable solutions, that they can compare with their programs' results. The importance of domain knowledge to help one reason about the program when debugging is clear from the work of Katz and Anderson (1987); instructors may need to consider the importance of designing programming assignments relevant to their particular students. Gugerty and Olsen (1986) showed the importance of not inserting new bugs into code while debugging. Training students to think in a hypothesis testing mode, including backing up from the change if it does not fix the bug, should be an important aspect of debugging instruction.

Spohrer and Soloway (1986a) found that misuse of logical operations and confusion about the order of arithmetic operations occurred more frequently than errors based on language constructs. Along with overlearning activities for the language, students should be given opportunities to debug these and other common bugs.

6.4. Tools

Brusilovsky (1993) and Wilson (1987) found one-on-one human intervention efficacious. Constraints on faculty time are unlikely to make this a viable alternative. However, numerous tutoring tools and visualizations have been developed in attempts to replace human prompting. While intelligent tutors and program visualization are beyond the scope of this study, tool developers would do well to heed Robertson et al. (2004), who found that systems that interrupt students impede their ability to comprehend code and work productively.

7. Research directions

Additional empirical evidence to support or reinforce recommended approaches to teaching would offer valuable additions to the literature. In particular, updating and replicating those experiments reviewed in this paper in today's learning context would be beneficial, as so many of those experiments were carried out several decades ago. We discuss here specific debugging research recommendations in the light of current languages, paradigms, and pedagogies. We also offer suggestions for broader research questions and point out several areas related to debugging that have not yet been investigated.

Spohrer and Soloway (1986a) demonstrated that, for the students they studied, control structures were not a major source of errors. In the process they identified a host of more common non-construct-based errors. Do their findings ring true for students in introductory classes today? It seems reasonable to assume that the complexities of object-oriented programming may well contribute to new sources of errors not observed in previous non-object-oriented studies. It also seems possible that due to the inclusion of so much new material at the introductory level students today are likely to spend less time focusing on control structures than students in the past, suggesting that they may in fact have more difficulties with control structures than the evidence from 30 years ago would indicate.

More research is needed on how best to help students confront the superbug (Pea, 1986) and understand that computers read programs through a strictly mechanistic and interpretive process, the rules of which are fairly simple once understood. Is the notion of a superbug exacerbated when students write event-driven or object-oriented programs? Event handling and super class methods, which are executed seemingly by magic without explicitly being called, would seem to add an extra layer of complexity that can lead to confusion. Does giving students more opportunities to read and understand existing programs, as suggested by Ahmadzadeh et al. (2005), help them overcome these challenges and improve their programming and debugging skills?

Ko and Myers (2005) suggested that "environmental factors," such as the programming environment, language features and external interruptions, are partly responsible for students' difficulties. Investigations into the roles of environment, languages, and pedagogical tools in facilitating students' acquisition of debugging skills offer new research opportunities. Specific examples are provided in Table 4.

Table 4. Environmental factors.

Environmental factors	Tool
Environments	Continuously compiling IDE's
Languages	Procedural versus object-oriented
	Scripting languages
Pedagogical tools	UML
	Design patterns
	Roles of variables
	Pair-programming

Also worthy of exploration are debugging approaches that capitalize on pedagogical tools (e.g. the ''pair debugging'' protocol). Such approaches might integrate Wilson's (1987) Socratic approach.

Further investigation into expert debugging behaviors as well as the differences between novice and expert debugging appear merited. Do original findings about expert behavior (Gould, 1975) remain valid in more modern software development contexts? One type of expertise which has not been researched is the debugging behavior of computer science educators, who are experts at debugging novice programs.

Investigations of differences between effective and ineffective novices may shed more light on developments for novice instruction. Improvements in classifying ''good'' and ''weak'' debuggers, as well as further investigations into the links between programming and debugging ability (Ahmadzadeh et al., 2005), could provide critical benchmarks for assessing the effectiveness of new debugging pedagogies or interventions, thus leading to insights into moving students from being unsuccessful to successful debuggers.

Research relating debugging ability to psychological factors or cognitive abilities has only been indirectly explored through general investigations of programming success. Examples include learning styles (Goold & Rimmer, 2000; Thomas, Ratcliffe, Woodbury, & Jarman, 2002), visual tests of field independence (Mancy & Reid, 2004), and abstraction ability (Bennedsen & Caspersen, 2006). Connecting such abilities to debugging could offer insights for instructional approaches or exercises designed to hone related skills. For example, Bednarik and Tukiainen (2005) believed that tests using blurred code, which affected experts but not novices, implied that ''experts [were] probably processing much information through peripheral vision.'' This suggests tests of visual ability, such as field independence, may be linked to debugging ability. The relationships between debugging success and individual factors such as goal orientation, motivation, and other self theories (Dweck, 1999) have not been researched. Students' self theories, particularly perceptions of their own intelligence as being either malleable or fixed, have been correlated respectively with mastery-oriented and helpless responses in the face of challenging tasks (Dweck, 1999). Given that debugging presents challenges for many novices, exploring the influence of self theories on student approaches to debugging appears worthwhile.

8. Brief summary of the literature

We hope this survey of the debugging literature helps computer science educators and researchers continue to seek answers to these questions and inspires future contributions to this body of knowledge. We conclude with a table summarizing the literature discussed (Table 5).

Table 5. Summary of readings.

Author	Year	Findings
Why do bugs occur?		
Soloway, Ehrlich, & Bonar	1982	Goal/plan analysis
Soloway & Ehrlich	1984	Goal/plan analysis
Johnson & Soloway	1984	Goal/plan analysis
Spohrer, Soloway, & Pope	1989	Goal/plan analysis
Bonar & Soloway	1985	Inappropriately applying natural language
Spohrer & Soloway	1986a	Analyzed high frequency bugs
Perkins & Martin	1986	Fragile knowledge
Pea	1986	Conceptual bugs/the superbug
Ko & Myers	2005	Framework; chains of cognitive breakdowns
What types of bugs occur?		
IEEE	1994	List of bug classifications
Johnson, Soloway, Cutler, & Draper	1983	Identification, categorization; Bug Catalog I
Spohrer et al.	1985	Categorization in order to understand novice problems; Bug Catalogs II, III & IV
Ahmadzadeh, Elliman, & Higgins	2005	Categorized six common semantic errors via compiler detection
Ford & Teorey	2002	Categorization from observation of novices
Hristova, Misra, Rutter, & Mercuri	2003	Surveyed faculty, TA's and students
Knuth	1989	Categorized bugs while writing TeX
Spohrer & Soloway	1986b	A few bugs account for most mistakes
Robins, Haden, & Garner	2006	Problems observed in object-based Java programming lab sessions
Hanks	2007	Pair programmers encounter similar problems but require less intervention
Gould	1975	Which bugs are hardest to detect?
What is the debugging process?		
Ben-Ari	1998	Constructivist approach
What types of knowledge are aids in debugging?		
Gould & Drongowski	1974	Sample output data used by experts during debugging
Gould	1975	Meaningful variable names, comments, application domain knowledge important
Katz & Anderson	1987	Output, program comprehension important
Carver & Risinger	1987	Output important
Gugerty & Olson	1986	Program comprehension important
Nanja & Cook	1987	Program comprehension important
Ducassé & Emde	1988	Identified types of bug locating knowledge
Ahmadzadeh, Elliman, & Higgins	2005	Good vs. weak debuggers; program comprehension important
Grant & Sackman	1967	Good at one programming activity, good at others
What strategies are employed in debugging?		
Gould & Drongowski	1974	Tactics, clues, location of bug or hypothesis
Gould	1975	Tactics, clues, location of bug or hypothesis
Vessey	1985	Debugging episodes and strategy paths
Katz & Anderson	1987	Debugging as troubleshooting – understand, test, locate, repair
Ducassé & Emde	1988	Global strategies - filtering, computational equivalence, well-formedness, stereotypical errors

(*continued*)

Table 5. (*Continued*).

Author	Year	Findings
Tubaishat	2001	Shallow and deep reasoning; BUG-DOCTOR
Carver & Risinger	1987	Debugging can be explicitly taught
How do novices and experts differ?		
Jeffries	1982	Experts have superior program comprehension
Vessey	1985	Experts vs. novices; ability to chunk
Gugerty & Olson	1986	Experts vs. novices; program comprehension ability
Nanja & Cook	1987	Experts read, comprehend, correct groups of errors; novices modify more code and rewrite functions
Fix, Wiedenbeck, & Scholtz	1993	Experts hierarchically structure representations, make connections, recognize recurring patterns
Bednarik & Tukiainen	2005	Code blurring affects experts, not novices
How can we improve the learning and teaching of debugging?		
Kessler & Anderson	1986	Debugging must be taught
Carver & Risinger	1987	Explicit debugging instruction is effective; flowchart of debugging process
Chmiel & Loui	2004	Explicit debugging instruction is effective; debugging exercises, logs, collaborative assignments
Brusilovsky	1993	Escalating human debugging assistance
Wilson	1987	One-on-one Socratic debugging assistance
Robertson et al.	2004	Immediate interruption styles are detrimental to short-term memory; negotiated interruptions more effective
Tubaishat	2001	Shallow and deep reasoning,
Wilcox et al.	1997	Continuous visual feedback not a panacea
Thomas, Ratcliffe, & Thomasson	2004	Students do not use diagrams

Acknowledgements

We thank Raymond Lister who participated in some early discussions of the debugging literature prior to this review being written. 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

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). Novice programmers: An analysis of patterns of debugging among novice computer science students. *Inroads*, *37*(3), 84–88.

Barnes, D., & Kölling, M. (2002). *Objects first with Java: A practical introduction using BlueJ*. Upper Saddle River, NJ: Prentice-Hall.

Bednarik, R., & Tukiainen, M. (2005). Effects of display blurring on the behavior of novices and experts during program debugging. In *CHI'05 Extended abstracts on human factors in computing systems* (pp. 1204–1207). Portland, OR: ACM Press.

Ben-Ari, M. (1998). Constructivism in computer science education. *Inroads*, *30*(3), 257–261.

Bennedsen, J., & Caspersen, M.E. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? *Inroads*, *38*(2), 39–43.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, *1*(2), 133–161.

Bransford, J., Brown, A. & Cocking, R., (Eds.). (2000). *How people learn: Brain, mind, experience, and school*. Washington, DC: National Academy Press.

Brusilovsky, P. (1993). Program visualization as a debugging tool for novices. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel & T. White (Eds.), *Proceedings of INTERACT '93 and CHI '93 conference companion on human factors in computing systems* (pp. 29–30). NewYork: ACM Press.

Carver, S., & Risinger, S. (1987). Improving children's debugging skills. In G. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical studies of programmers: Second Workshop* (pp. 147–171). Norwood, NJ: Ablex.

Chmiel, R., & Loui, M. (2004). Debugging: From novice to expert. *Inroads*, *36*(1), 17–21.

Ducassé, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In W. Schäfer & Pere Botella (Eds.), *Proceedings of the 10th international conference on software engineering* (pp. 162–171). Singapore: IEEE Computer Society Press.

Dweck, C.S. (1999). *Self-theories: Their role in motivation, personality and development*. Philadelphia, PA: Psychology Press.

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. In P. Bauersfeld, J. Bennett & G. Lynch (Eds.), *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 74–79). New York: ACM Press.

Ford, A., & Teorey, T. (2002). *Practical debugging in C++*. Upper Saddle River, NJ: Prentice-Hall.

Goold, A., & Rimmer, R. (2000). Factors affecting performance in first-year computing. *Inroads*, *32*(2), 39–43.

Gould, J. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man–Machine Studies*, *7*(1), 151–182.

Gould, J., & Drongowski, P. (1974). An exploratory study of computer program debugging. *Human Factors*, *16*, 258–277.

Grant, E., & Sackman, H. (1967). An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, *8*(1), 33–48.

Gugerty, L., & Olson, G. (1986). Debugging by skilled and novice programmers. In M. Mantei & P. Orbeton (Eds.), *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 171–174). New York: ACM Press.

Hanks, B. (2007). Problems encountered by novice pair programmers. In S. Fincher, M. Guzdial & R. Anderson (Eds.), *Proceedings of the 3rd international computing education research workshop* (pp. 159–164). New York: ACM Press.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *Inroads*, *35*(1), 153–156.

IEEE (1994). *IEEE standard 1044–1993: Standard classification for software anomalies*. Retrieved November 21, 2007 from http://standards.ieee.org/reading/ieee/std_public/description/se/1044–1993_desc.html

Jeffries, R. (1982, March). *A comparison of debugging behavior of expert and novice programmers*. Paper presented at the American Education Research Association Annual Meeting, New York.

Johnson, W., & Soloway, E. (1984). PROUST: Knowledge-based program understanding. In T. Straeter, W. Howden & J. Rault (Eds.), *Proceedings of the 7th international conference on software engineering* (pp. 369–380). Piscataway, NJ: IEEE Press.

Johnson, W., Soloway, E., Cutler, B., & Draper, S. (1983). *Bug catalogue: I* (Technical Report No. 286). New Haven, CT: Yale University, Department of Computer Science.

Katz, I., & Anderson, J. (1987). Debugging: An analysis of bug location strategies. *Human–Computer Interaction*, *3*(4), 351–399.

Kessler, C., & Anderson, J. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 198–212). Norwood, NJ: Ablex.

Knuth, D. (1989). The errors of TEX. *Software – Practice and Experience*, *19*(7), 607–685.

Ko, A., & Myers, B. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, *16*, 41–84.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *Inroads*, *36*(4), 119–150.

Mancy, R., & Reid, N. (2004). Aspects of cognitive style and programming. In E. Dunican & T. Green (Eds.), *Proceedings of the 16th workshop of the psychology of programming interest group*, Carlow, Ireland. Retrieved May 20, 2008, from http://www.ppig.org/workshops/16th-programme.html

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Inroads*, *33*(4), 125–140.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM*, *49*(8), 90–95.

Metzger, R. (2004). *Debugging by thinking: A multidisciplinary approach*. Burlington, MA: Elsevier Digital Press.

Nanja, M., & Cook, C.R. (1987). An analysis of the on-line debugging process. In G. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 172–184). Norwood, NJ: Ablex.

Pea, R.D. (1986). Language-independent conceptual bugs in novice programming. *Journal of Educational Computing Research*, *21*, 25–36.

Perkins, D., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 213–229). Norwood, NJ: Ablex.

Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L., et al. (2004). Impact of interruption style on end-user debugging. In E. Dykstra-Erickson & M. Tscheligi (Eds.), *Proceedings of the 2004 conference on human factors in computing systems* (pp. 287–294). New York: ACM Press.

Robins, A., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian computing education conference* (pp. 165–173). Hobart: Australasian Computer Society.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *10*(5), 595–609.

Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. In *Proceedings of the 1982 conference on human factors in computing systems* (pp. 52–57). Gaithersburg, MD: ACM Press.

Spohrer, J., & Soloway, E. (1986a). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 230–251). Norwood, NJ: Ablex.

Spohrer, J., & Soloway, E. (1986b). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632.

Spohrer, J., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 356–399). Hillsdale, NJ: Erlbaum.

Spohrer, J., Pope, E., Lipman, M., Sack, W., Freiman, S., Litman, D., et al. (1985). *Bug catalogue: II, III, IV*. (Technical Report No. 386). New Haven, CT: Yale University, Department of Computer Science.

Thomas, L., Ratcliffe, M., & Thomasson, B. (2004). Scaffolding with object diagrams in first year programming classes: Some unexpected results. *Inroads*, *36*(1), 250–254.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. *Inroads*, *34*(1), 33–37.

Tubaishat, A. (2001). A knowledge base for program debugging. In *Proceedings of the international conference on computer systems and applications* (pp. 321–327). Beirut: IEEE Press.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man–Machine Studies*, *23*, 459–494.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man–Machine Studies*, *23*, 383–390.

Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., & Cook, C. (1997). Does continuous visual feedback aid debugging in direct-manipulation programming systems? In S. Pemberton (Ed.), *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 258–265). New York: ACM Press.

Wilson, J. (1987). A Socratic approach to helping novice programmers debug programs. *SIGCSE Bulletin*, *19*(1), 179–182.

Zeller, A. (2006). *Why programs fail: A systematic guide to debugging*. San Francisco, CA: Morgan Kaufmann.