ARTICLE IN PRESS

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Visual Languages and Computing
▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of Visual Languages & Computing

www.elsevier.com/locate/jvlc

A framework and methodology for studying the causes of software errors in programming systems

Andrew J. Ko*, Brad A. Myers

Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA

Received 1 January 2004; received in revised form 1 July 2004; accepted 1 August 2004

Abstract

An essential aspect of programmers' work is the correctness of their code. This makes current HCI techniques ill-suited to analyze and design the programming systems that programmers use everyday, since these techniques focus more on problems with learnability and efficiency of use, and less on error-proneness. We propose a framework and methodology that focuses specifically on errors by supporting the description and identification of the causes of software errors in terms of chains of cognitive breakdowns. The framework is based on both old and new studies of programming, as well as general research on the mechanisms of human error. Our experiences using the framework and methodology to study the Alice programming system have directly inspired the design of several new programming tools and interfaces. This includes the Whyline debugging interface, which we have shown to reduce debugging time by a factor of 8 and help programmers get 40% further through their tasks. We discuss the framework's and methodology's implications for programming system design, software engineering, and the psychology of programming.

*Corresponding author. Tel.: +1 412 268 1266; fax: +1 412 401 0042.
E-mail address: ajko@cmu.edu (A.J. Ko).

1045-926X/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2004.08.003

1. Introduction

> "Human fallibility, like gravity, weather and terrain, is just another foreseeable hazard… The issue is not why an error occurred, but how it failed to be corrected. We cannot change the human condition, but we can change the conditions under which people work."
>
> James Reason, Managing the Risks of Organizational Accidents [1]

In 2002, The National Institute of Standards and Technology published a study of major U.S. software engineering industries, finding that software engineers spend an average of 70–80% of their time testing and debugging, with the average bug taking 17.4 hours to fix. The study estimated that such testing and debugging costs the US economy over $50 billion annually [2]. One reason for these immense costs is that as software systems become increasingly large and complex, the difficulty of detecting, diagnosing, and repairing software problems has also increased. Because this trend shows no sign of slowing, there is considerable interest in designing programming systems that can demonstrably prevent errors, and better help programmers find, diagnose and repair the unprevented errors.

Unfortunately, the design and evaluation of such "error-robust" programming systems still poses a significant challenge to HCI research. Most techniques that have been proposed for evaluating computerized systems, such as GOMS [3] and Cognitive Walkthroughs [4], have focused on low-level details of interaction, bottlenecks in learnability and performance, and the close inspection of simple tasks. In programming activity, however, even "simple" tasks are complex, and productivity bottlenecks are more often in *repairing* errors than in learning to avoid them. Even with the more design-oriented HCI techniques, understanding a programming system's error-proneness has been something of a descriptive dilemma. Nielsen's Heuristic Evaluation suggests little more than to prevent user errors by finding common error situations [5]. The Cognitive Dimensions of Notations framework [6], though applied to numerous programming systems [7,8], characterizes *error-proneness* simply as "the degree to which a notation invites mistakes." In this paper, we offer an alternative technique, specifically designed to objectively analyze a programming system's influence on errors. We integrate several of our recent studies with three strands of prior research:

- Past classifications of common programming difficulties;
- Studies of the cognitive difficulties of programming; and
- Research on the general cognitive mechanisms of human error.

From this research, we derive a framework for describing *chains of cognitive breakdowns* that lead to error, and a methodology for sampling these chains by observing programmers' interaction with a programming system. We hope that these contributions will not only be valuable tools for improving existing programming languages and environments, both visual and textual, but also for guiding the design of new error-robust languages, environments, and interactive visualizations.

This paper is divided into six parts. In the next section, we review classifications of common programming difficulties, studies of programming that suggest several causes of error, and research on the general mechanisms of human error. In Section 3, we describe our framework in detail and in Section 4 we describe an empirical methodology for using the framework to study a programming system's error-proneness. In Section 5, we describe our experiences using the framework and methodology to analyze the Alice programming system [10]. We end in Section 6 with a discussion of the strengths and applicability of our framework and methodology to programming system design, software engineering, and the psychology of programming.

2. Definitions, classifications and causes

In this section, we review three strands of research: classifications of common programming difficulties, studies of cognitive difficulties in programming, and research on the general mechanisms of human error. To help frame our discussion, let us first define some relevant terminology.

2.1. Terminology

If the goal of software engineering is to build a product that meets a particular need, the *correctness* of a software system can be defined relative to interpretations of this need:

- General expectations of the software's behavior and functionality;
- A software designer's interpretation of these expectations, known as *requirement specifications*;
- A software architect's formal and informal interpretations of the requirement specifications, known as *design specifications*;
- A programmer's understanding, or mental model, of design specifications.

Because we are interested in how a programming system can help improve correctness, we define correctness relative to *design specifications*. While there can certainly be problems with design specifications, as well as requirements, such problems are typically outside the influence of programming systems.

Given this definition of correctness, we define three terms: *runtime failures*, *runtime faults*, *and software errors* (illustrated in Fig. 1). A *runtime failure* is an event that occurs when a program's behavior—often some form of visual or numerical program output—does not comply with the program's design specifications. A *runtime fault* is a machine state that may cause a runtime failure (e.g., a wrong value in a CPU register, branching to an invalid memory address, or a hardware interrupt that should not have been activated). A *software error* is a fragment of code that may cause a runtime fault during program execution. For example, software errors in

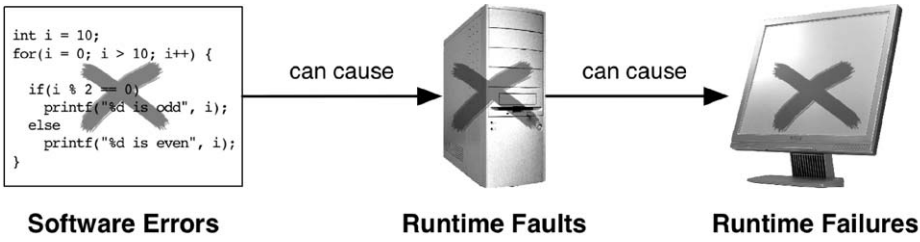

Fig. 1. The relationship between software errors in code, runtime faults during execution, and runtime failures in program behavior. These images will be used to represent these three concepts throughout this paper.

loops include a missing increment statement, a leftover "break" command from a debugging session, or a conditional expression that always evaluates to true. It is important to note that while a runtime failure guarantees that one or more runtime faults have occurred, and a runtime fault guarantees that one or more software errors exist, software errors do not *always* cause runtime faults, and runtime faults do not *always* cause runtime failures. Also note that under our definition, a single change to the design specifications can introduce an arbitrary number of software errors.

Using these definitions, a number of other terms can be clarified. A *bug* is an amalgam of one or more software errors, runtime faults, and runtime failures. For example, a programmer can refer to a software error as a bug, as in "Oh, there's the bug on line 43," as a runtime failure, as in "Oh, don't worry about that. It's just a bug…" or even as all three, as in "I fixed four bugs today." *Debugging* involves determining what runtime faults led to a runtime failure, determining what software errors were responsible for those runtime faults, and modifying the code to prevent the runtime faults from occurring. *Testing* involves searching for runtime failures and recording information about runtime faults to aid in debugging.

A *programming system* is a set of components (e.g., editors, debuggers, compilers, and documentation), each with (1) a *user interface*; (2) some set of *information*, such as program code or runtime data, which the programmer views and manipulates via the user interface; and (3) a *notation* in which the information is represented. We illustrate some common programming system components in the rows of Fig. 2. The figure is read from left to right; for example, a programmer *uses* diagrams and printed text *to view and manipulate* specifications, *which are represented in* natural language, UML, or some other notation; a programmer *uses* an editor *to view and manipulate* code, *which is represented in terms of* some programming language; a programmer *uses* a debugger *to view and manipulate* a machine's behavior, *which is in terms of* stack traces, registers, and memory; a programmer *uses* an output device *to view* a program's behavior, *which is often represented as* graphics, text, animation, and sound, etc.

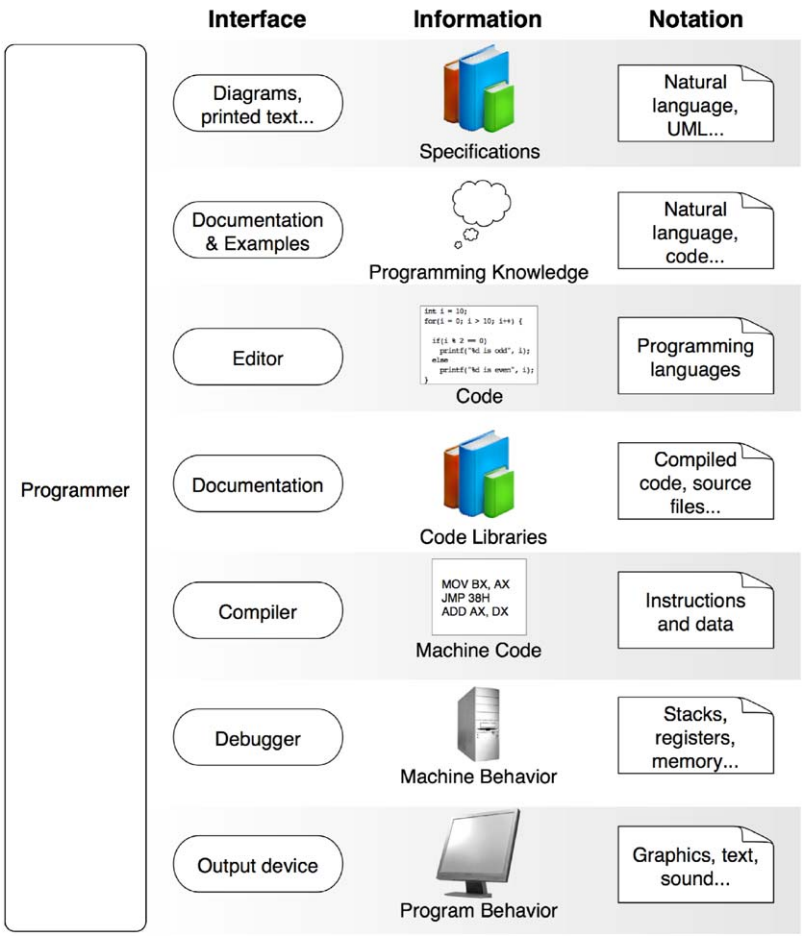

Fig. 2. A programming system has several components (in rows), each with an interface, information, and notation (in columns). The diagram is read from left to right, as in "Programmers *use* interface X *to view and manipulate* information Y, *which is represented in* notation Z."

2.2. Classifications of common programming difficulties

Prior work on classifying common programming difficulties—summarized chronologically in Table 1—has been reasonably successful in motivating novel and effective tools for finding, understanding and repairing software errors. For example, in the early 1980s, the Lisp Tutor drew heavily from analyses of novices' software errors [11], and nearly approached the effectiveness of a human tutor. More recently, the testing and debugging features of the Forms/3 visual spreadsheet language [12] were largely motivated by studies of the type and prevalence of spreadsheet errors [13].

Table 1
Studies classifying "bugs", "errors" and "problems" in various languages, expertise, and programming contexts

Study	Bug/Error/ Problem	Description	Author's comments
Gould [14] Novice Fortran	Assignment bug	Software errors in assigning variables' values	Requires understanding of behavior
	Iteration bug	Software errors in iteration algorithms	Requires understanding of language
	Array bug	Software errors in array index expressions	Requires understanding of language
Eisenberg [15] Novice APL	Visual bug	Grouping related parts of expression	
	Naive bug	Iteration instead of parallel processing	'…need to think step-by-step'
	Logical bug	Omitting or misusing logical connectives	
	Dummy bug	Experience with other languages interfering	'…seem to be syntax oversights'
	Inventive bug	Inventing syntax	
	Illiteracy bug	Difficulties with order of operations	
	Gestalt bug	Unforeseen side effects of commands	'…failure to see the whole picture'
Johnson et al. [16] Novice Pascal	Missing	Omitting required program element	Software errors have context: input/output, declaration, initialization and update of variables, conditionals, and scope delimiters.
	Spurious	Unnecessary program element	
	Misplaced	Required program element in wrong place	
	Malformed	Incorrect program element in right place	
Spohrer and Soloway [17] Novice Basic	Data-type inconsistency	Misunderstanding data types	'All bugs are not created equal. Some occur over and over again in many novice programs, while others are more rare…Most bugs result because novices misunderstand the semantics of some particular programming language construct.'
	Natural language	Applying natural language semantics to code	
	Human-interpreter	Assuming computer interprets code similarly	
	Negation & whole-part	Difficulties constructing Boolean expressions	
	Duplicate tail-digit	Incorrectly typing constant values	

Table 1 (*continued*)

Study	Bug/Error/ Problem	Description	Author's comments
	Knowledge interference	Domain knowledge interfering w/constants	
	Coincidental ordering	Malformed statements produce correct output	
	Boundary	Unanticipated problems with extreme values	
	Plan dependency	Unexpected dependencies in program	
	Expectation/ interpretation	Misunderstanding problem specification	
Knuth [18] While writing TeX in SAIL and Pascal	Algorithm awry	Improperly implemented algorithms	'proved…incorrect or inadequate'
	Blunder or botch	Accidentally writing code not to specifications	'not…enough brainpower'
	Data structure debacle	Software errors in using data structures	'did not preserve…invariants'
	Forgotten function	Missing implementation	'I did not remember everything'
	Language liability	Misunderstanding language/ environment	
	Module mismatch	Imperfectly knowing specification	'I forgot the conventions I had built'
	Robustness	Not handling erroneous input	'tried to make the code bullet-proof''
	Surprise scenario	Unforeseen interactions in program elements	'forced me to change my ideas'
	Trivial typos	Incorrect syntax, reference, etc.	'my original pencil draft was correct'
Eisenstadt [19] Industry experts COBOL, Pascal, Fortran, C	Clobbered memory	Overwriting memory, subscript out of bounds	Also identified why software errors were difficult to find: cause/effect chasm; tools inapplicable; failure did not actually happen; faulty knowledge of specs; ''spaghetti'' code.
	Vendor problems	Buggy compilers, faulty hardware	
	Design logic	Unanticipated case, wrong algorithm	
	Initialization	Erroneous type or initialization of variables	
	Variable	Wrong variable or operator used	
	Lexical bugs	Bad parse or ambiguous syntax	
	Language	Misunderstandings of language semantics	

Table 1 (*continued*)

Study	Bug/Error/ Problem	Description	Author's comments
Panko [13] Novice Excel	Omission error	Facts to be put into code, but are omitted	Quantitative errors: "errors that lead to an incorrect, bottom line value"
	Logic error	Incorrect or incorrectly implemented algorithm	
	Mechanical error	Typing wrong number; pointing to wrong cell	
	Overload error	Working memory unable to finish without error	Qualitative errors: "design errors and other problems that lead to quantitative errors in the future"
	Strong but wrong error	Functional fixedness (a fixed mindset)	
	Translation error	Misreading of specification	

Despite the successful use of these classifications, in hindsight it is clear that the classifications do not actually classify *software errors*, but rather, the complex relationships between software errors, runtime faults, runtime failures, and cognitive failures. Nevertheless, in analyzing these classifications, four salient aspects of software errors emerge.

The first is a software error's *surface qualities*: the particular syntactic or notational anomalies that make a code fragment incorrect. Eisenberg's *dummy bug* is a class of syntax oversights; Knuth's *trivial typos* and Panko's *mechanical errors* simply describe unintended text in a program; Gould identifies particular surface qualities of erroneous assignment statements and array references in his study of Fortran. Clearly, the surface qualities of software errors are greatly influenced by the language syntax. While it may seem that these qualities have little to do with the actual cause of software errors, the fact that they are common enough to warrant their own category suggests that language syntax can be a cause of software errors on its own.

Other categories allude to several *cognitive* causes of software errors. For example, Eisenberg's *inventive bug*, Spohrer and Soloway's *data-type inconsistency*, and Johnson's *misplaced* and *malformed* categories all refer to programmers' lack of *knowledge* about language syntax, control constructs, data types, and other programming concepts. Knuth's *forgotten* function category and Eisenstadt's *variable bugs* suggest *attentional* issues such forgetting or a lack of vigilance. Eisenstadt's *design logic bugs* and Knuth's *surprise scenario* category indicate *strategic* issues, referring to problems like unforeseen code interactions or poorly designed algorithms.

A third aspect of software errors is the *programming activity* in which the cause of the software error occurred. For example, Knuth's *mismatch between modules* bugs and Spohrer and Soloway's *expectation and interpretation* problems all occur in specification activities, in which the programmer's invalid or inadequate

Table 2
Actions performed during programming activity

Action	Examples of the action in programming activity
Creating	Writing code, or creating design and requirement specifications
Reusing	Reusing example code, copying and adapting existing code
Modifying	Modifying code or changing specifications
Designing	Considering various software architectures, data types, algorithms, etc.
Exploring	Searching for code, documentation, runtime data
Understanding	Comprehending a specification, an algorithm, a comment, runtime behavior, etc.

comprehension of design specifications later led to software errors. Spohrer and Soloway's *plan dependency problem* occurs during algorithm design activities, in which unforeseen interactions eventually led to software errors.

A fourth and final aspect of software errors is the type of *action* that led to the error. The classifications suggest six types of programming actions, which we list in Table 2 with examples. For instance, programmers can introduce software errors when *creating* code, but the creation of specifications can also predispose software errors (as in Spohrer and Soloway's *expectation/interpretation* problems). Programmers also *reuse* code, *modify* specifications and code, *design* software architectures and algorithms and *explore* code and runtime data. The classifications also blame the *understanding* of specifications, data structures, and language constructs for several types of software errors.

While these classifications go a long way in conveying the scope and complexity of several aspects of software errors, they only go so far in relating these aspects causally. For example, what looks like an erroneously coded algorithm on the surface may have been caused by an invalid understanding of the specifications, a lack of experience with a language construct, misleading information from a debugging session, or simply momentary inattention. Each possible cause motivates entirely different interventions.

2.3. Human error in programming activity

To fully understand how the interaction between a programmer and a programming system can lead to software errors, we need a more general discussion of the underlying cognitive mechanisms of human error. James Reason's *Human Error* [20] provides a solid foundation for this discussion. In this section, we adapt two aspects of his research to the domain of programming: (1) a systemic view of the causes of failure, and (2) a brief catalog of common failures in human cognition.

2.3.1. Systemic causes of failure

Thus far we have considered what Reason refers to as *active errors*: errors whose effects are felt almost immediately, such as syntax errors that prevent successful

compilation or invalid algorithms. Reason also defines *latent errors*, ''whose adverse consequences may lie dormant within the system for a long time, only becoming evident when they combine with other factors to breach the system's defenses'' [20]. The fundamental idea is that complex systems have several functional layers, each with potential latent errors that predispose failure, but also with a set of defenses that prevent latent errors from becoming active. From this perspective, failures are ultimately due to a causal chain of failures both within and between layers of a system.

We apply these ideas to software engineering in Fig. 3. In the figure, we portray four layers, each with its own type of latent errors and defenses. On the left, we see that specifications act as high-level defenses against software errors, but if they are ambiguous, incomplete, or incorrect, they may predispose programmers to misunderstandings about a software system's true requirements. By improving software engineering practices, there will be fewer latent errors in design specifications, which will prevent programmers' invalid or incomplete understanding of specifications. Programmers, the next layer in Fig. 3, have knowledge, attention, and expertise to defend against software errors. However, programmers are also prone to *cognitive breakdowns* in these defenses, which predispose software errors. We will discuss these breakdowns in detail in the next section. The third layer in Fig. 3, the programming system, consists of several components (compilers, libraries, languages, environments, etc.). Each has a set of defenses against software errors, but also a set of latent usability issues that predispose the programmer to cognitive breakdowns, and thus software errors. For example, compilers defend against syntax errors, but in displaying confusing error messages, may misguide programmers in

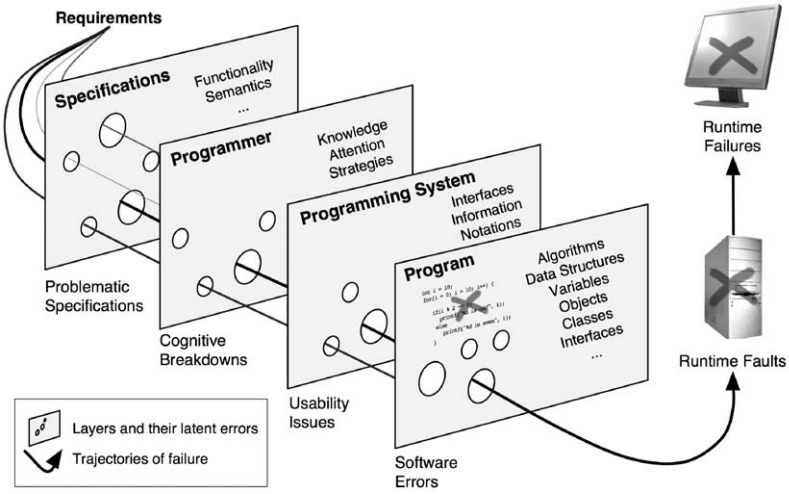

Fig. 3. Dynamics of software error production, based on Reason's systemic view of failure. Each layer has latent errors (the holes), predisposing certain types of failures. Layers also have defenses against failures (where there are no holes). Several layers of failure must go unchecked before software errors are introduced into code.

correctly repairing the syntax errors. The last layer, the program, has the latent errors we know as software errors, which can eventually lead to a program's runtime failure.

It is important to note that latent errors in these layers only become active in particular circumstances. Just as a program may only fail with particular input and in particular states, programming systems, programmers, and specifications may only fail in particular situations.

2.3.2. Skill, rule, and knowledge breakdowns

Within the broad view of software errors portrayed in Fig. 3, we now focus on the programmer's latent errors—what we will call *cognitive breakdowns*—and how the programming system might be involved in predisposing these cognitive breakdowns. Reason's central thesis about human behavior is that in any given context, individuals will behave in the same way they have in the past in that context. Under most circumstances, these "default" behaviors are sufficient; however, under exceptional or novel circumstances, they may lead to error. In programming, this means that when solving problems, programmers tend to prefer programming strategies that have been successful in the past. These default strategies are usually successful, but they sometimes break down—hence the term *cognitive breakdowns*—and lead to software errors.

In order to clarify the sources of these breakdowns, Reason discusses three general types of cognitive activity, each prone to certain types of cognitive breakdowns. The most proceduralized of the three, *skill-based* activity, usually fails because of a lack of attention given to performing routine, skillful patterns of actions. *Rule-based* activity, which is driven by learned expertise, usually fails because the wrong rule is chosen, or the rule is inherently bad. *Knowledge-based* activity, centered on conscious, deliberate problem solving, suffers from cognitive limitations and biases inherent in human cognition. We will discuss all three types of cognitive activity and their accompanying breakdowns in detail.

Skill-based activities are routine and proceduralized, where the focus of attention is on something other than the task at hand. Some skill-based activities in programming include typing a text string, opening a source file with a file open dialog, or compiling a program by pressing a button in the programming environment. These are practiced and routine tasks that can be left in "auto-pilot" while a programmer attends to more problem-oriented matters. An important characteristic of skill-based activities is that because attention is focused *internally* on problem solving and not *externally* on performing the routine action, programmers may not notice important changes in the external environment.

Table 3 lists Reason's two categories of skill breakdowns. *Inattention* breakdowns are a failure to pay attention to performing routine actions at critical times. For example, imagine a programmer finishing the end of a *for loop* header when the fire alarm goes off in his office. When he returns to the loop after the interruption, he fails to complete the increment statement, introducing a software error. Inattention breakdowns may also occur because of the intrusion of *strong habits*. For example, consider a programmer who tends to save modifications to a source file after every

Table 3
Types of skill breakdowns, adapted from Reason [20]. The → means "causes"

Inattention	Type	Events resulting in breakdown
Failure to attend to a routine action at a critical time causes forgotten actions, forgotten goals, or inappropriate actions.	Strong habit intrusion	In the middle of a sequence of actions → no attentional check → contextually frequent action is taken instead of intended action
	Interruptions	External event → no attentional check → action skipped or goal forgotten
	Delayed action	Intention to depart from routine activity → no attentional check between intention and action → forgotten goal
	Exceptional stimuli	Unusual or unexpected stimuli → stimuli overlooked → appropriate action not taken
	Interleaving	Concurrent, similar action sequences → no attentional check → actions interleaved
Overattention	**Type**	**Events resulting in breakdown**
Attending to routine action causes false assumption about progress of action.	Omission	Attentional check in the middle of routine actions → assumption that actions are already completed → action skipped
	Repetition	Attentional check in the middle of routine actions → assumption that actions are not completed → action repeated

change so that important modifications are not lost. At one point, he deletes a large block of code he thinks is unnecessary, but immediately after, realizes he needed the code after all. Unfortunately, his strong habit of saving every change has already intruded, and he loses the code permanently (a good motivation for sophisticated undo mechanisms in programming environments).

Overattention breakdowns occur when attending to routine actions that would have been better left to "auto-pilot." For example, imagine a programmer has copied and pasted a block of code and is quickly coercing each reference to a contextually appropriate variable. While planning his next goal in his head, he notices that he has not been paying attention and interrupts his "auto-pilot," accidentally looking two lines down from where he actually was. He falsely assumes that the statements above were already coerced, leaving several invalid references in his code.

Rule-based activities involve the use of cognitive rules for acting in certain contexts. These rules consist of some *condition*, which checks for some pattern of *signs* in the current context. If the current context matches the condition, then corresponding *actions* are performed. For example, expert C programmers frequently employ the rule, "If some operation needs to be performed on the elements of a list, type *for* (*int* $i = some\_initial\_value$; $i < some\_terminating\_value$; $i++$), choose the initial and terminating values, then perform the operation." These

rules are much like the concept of programming plans [21], which are thought to underlie the development of programming expertise [22].

Table 4 lists Reason's two categories of rule breakdowns, the first of which is *wrong rule*. Because rules are influenced by prior experience, they make implicit predictions about the future state of the world. These predictions of when and how the world will change are sometimes incorrect, and thus a rule that is perfectly reasonable in one context may be selected in an inappropriate context. For example, one common breakdown in Visual Basic.NET is that programmers will use the "+" operator to add numeric values, not realizing that the values are represented as strings, and so the strings are concatenated instead. Under normal circumstances, use of the "+" operator to add numbers is a perfectly reasonable rule; however, because there were no distinguishing signs of the variables' types in the code, it was applied inappropriately.

Empirical studies of programming have reliably demonstrated many other types of *wrong rule* breakdowns. For example, Davies' framework of knowledge

Table 4
Types of rule breakdowns, adapted from Reason [20]

Wrong rule	Type	Events resulting in breakdown
Use of a rule that is successful in most contexts, but not all.	Problematic signs	Ambiguous or hidden signs → conditions evaluated with insufficient info → wrong rule chosen → inappropriate action
	Information overload	Too many signs → important signs missed → wrong rule chosen → inappropriate action
	Favored rules	Previously successful rules are favored → wrong rule chosen → inappropriate action
	Favored signs	Previously useful signs are favored → exceptional signs not given enough weight → wrong rule chosen → inappropriate action
	Rigidity	Familiar, situationally inappropriate rules preferred over unfamiliar, situationally appropriate rules → wrong rule chosen → inappropriate action
Bad rule	**Type**	**Events resulting in breakdown**
Use of a rule with problematic conditions or actions.	Incomplete encoding	Some properties of problem space are not encoded → rule conditions are immature → inappropriate action
	Inaccurate encoding	Properties of problem space encoded inaccurately → rule conditions are inaccurate → inappropriate action
	Exception proves rule	Inexperience → exceptional rule often inappropriate → inappropriate action
	Wrong action	Condition is right but action is wrong → inappropriate action

restructuring in the development of programming expertise suggests that a lack of training in structured programming can lead to the formation of rules appropriate for one level of program complexity, but inappropriate for higher levels of complexity [22]. For example, in Visual Basic, the rule "if some data needs to be used by multiple event-handlers, create a global variable on the form" is appropriate for forms with a small number of event-handlers, but quickly becomes unmanageable in programs with hundreds. Similarly, Shackelford studied the use of three types of Pascal *while* loops, finding that while most students had appropriate rules for choosing the type of loop for a problem, the same rules failed when applied to similar problems with additional complexities [23].

The second type of rule breakdown is the use of a *bad rule*: one with problematic conditions or actions. These rules come from learning difficulties, inexperience, or a lack of understanding about a particular program's semantics. For example, Perkins and Martin demonstrated that "fragile knowledge"—inadequate knowledge of programming concepts, algorithms, and data structures, or an inability to apply the appropriate knowledge or strategies—was to blame for most novice software errors when learning Pascal [17]. Not knowing the language syntax—in other words, not encoding or inaccurately encoding its properties—can lead to simple syntax errors, malformed Boolean logic, scoping problems, the omission of required constructs, and so on. An inadequate understanding of a sorting algorithm may cause a programmer to unintentionally sort a list in the wrong order. Von Mayrhauser and Vans illustrated that programmers who focused only on comprehending surface level features of a program (variable and method names, for example), and thus had an insufficient model of the program's runtime behavior, did far worse in a corrective maintenance task than those who focused on the program's runtime behavior [24].

In *knowledge-based* activities, Reason's last type of cognitive activity, the focus of attention is on forming plans and making high-level decisions based on one's knowledge of the problem space. In programming, knowledge-based activities include forming a hypothesis about what caused a runtime failure, or comprehending the runtime behavior of an algorithm. Because knowledge-based activities rely heavily on the interpretation and evaluation of models of the world (in programming, models of a program's semantics), they are considerably taxing on the limited resources of working memory. This results in the use of a number of cognitive "shortcuts" or biases, which can lead to cognitive breakdowns.

Table 5 describes these biases, and how they cause breakdowns in the strategies and plans that programmers form. One important bias is *bounded rationality* [25]: the idea that the problem spaces of complex problems are often too large to permit an exhaustive exploration, and thus problem solvers "satisfice" or explore "enough" of the problem space. Human cognition uses a number of heuristics to choose which information to consider [20]: (1) evaluate information that is easy to evaluate (*selectivity*); (2) only evaluate as much as is necessary to form a plan of action (*biased reviewing*); (3) evaluate information that is easily accessible in the world or in the head (*availability*).

Because of the complexity of programming activity, bounded rationality shows up in many programming tasks. For example, Vessey argues that debugging is difficult

Table 5
Types of knowledge breakdowns, adapted from Reason [20]

Bounded rationality	Type	Events resulting in breakdown
Problem space is too large to explore because working memory is limited and costly.	Selectivity	Psychologically salient, rather than logically important task information is attended to → biased knowledge
	Biased reviewing	Tendency to believe that all possible courses of action have been considered, when in fact very few have been considered → suboptimal strategy
	Availability	Undue weight is given to facts that come readily to mind → facts that are not present are easily ignored → biased knowledge
Faulty models of problem space	**Type**	**Events resulting in breakdown**
Formation and evaluation of knowledge leads to incomplete or inaccurate models of problem space.	Simplified causality	Judged by perceived similarity between cause and effect → knowledge of outcome increases perceived likelihood → invalid knowledge of causation
	Illusory correlation	Tendency to assume events are correlated and develop rationalizations to support the belief → invalid model of causality
	Overconfidence	False belief in correctness and completeness of knowledge, especially after completion of elaborate, difficult tasks → invalid, inadequate knowledge
	Confirmation bias	Preliminary hypotheses based on impoverished data interfere with later interpretation of more abundant data → invalid, inadequate hypotheses

because the range of possible software errors causing a runtime failure is highly unconstrained and further complicated by that fact that multiple independent or interacting software errors may be to blame [26]. Gilmore points out that, because of their limited cognitive resources, programmers generally only consider a few hypotheses of what software errors caused the failure, and usually choose an incorrect hypothesis. This not only leads to difficulty in debugging, but often the introduction of further software errors due to incorrect hypotheses [27]. For example, in response to a program displaying an unsorted list because the sort procedure was not called, a programmer might instead decide the software error was an incorrect swap algorithm, and attempt to modify the already correct swap code.

The second type of knowledge breakdown is the use of a *faulty model of the problem space*. For example, human cognition tends to see *illusory correlations* between events, and even develops rationalizations to defend such beliefs in the face of more accurate observations (*confirmation bias*). These biases lead to oversimplified

or incorrect models of the problem space. Individuals also display *overconfidence*, giving undue faith to the correctness and completeness of their knowledge. This results in strategies that are based on incomplete analyses. For example, spreadsheet users exhibit so much overconfidence in their spreadsheets' formulas that a single test case is often enough to convince them of their spreadsheet's correctness [28]. Corritore and Wiedenbeck have shown that programmers' overconfidence in the correctness of their mental models of a program's semantics was often the cause of software errors in programmers' modifications [29].

3. A framework for studying the causes of software errors

As we have seen, the causes of software errors are rarely due to a programmer's cognitive failures alone: a myriad of environmental factors, such as hidden or ambiguous signs in a programming environment, unfortunately timed interruptions, or poorly conceived language constructs may also be involved. Thus, to truly support design, the programmer *and* the programming system should be considered together.

To this end, we propose a framework that combines aspects of both programming systems and human cognition:

1. Programmers perform three types of *programming activities*: specification activities (involving design and requirements specification), implementation activities (involving the manipulation of code), and runtime activities (involving testing and debugging) (Section 2.2).
2. Programmers perform six types of *actions* while interacting with a programming system's interfaces: design, creation, reuse, modification, understanding, and exploration (Section 2.2).
3. *Skill breakdowns*, *rule breakdowns*, and *knowledge breakdowns* occur as a result of the interaction between programmers' cognitive limitations and properties of the programming system and external environment (Section 2.3.2).

We combine these aspects into two central ideas:

1. *A cognitive breakdown* consists of four components: the *type* of breakdown, the *action* being performed when the breakdown occurs, the *interface* on which the action is performed, and the *information* that is being acted upon.
2. *Chains of cognitive breakdowns* are formed over the course of programming activity, often leading to the introduction of software errors.

These ideas map directly to elements in our framework, which we portray in Fig. 4. The three grey regions, stacked vertically, denote specification, implementation, and runtime activities. The four columns contain various types of the four components of a breakdown. Breakdowns are read from left to right in the figure, with '[ ]' meaning "choose one within the brackets." For example, in specification activities, a single breakdown consists of one of three types of breakdowns, one of

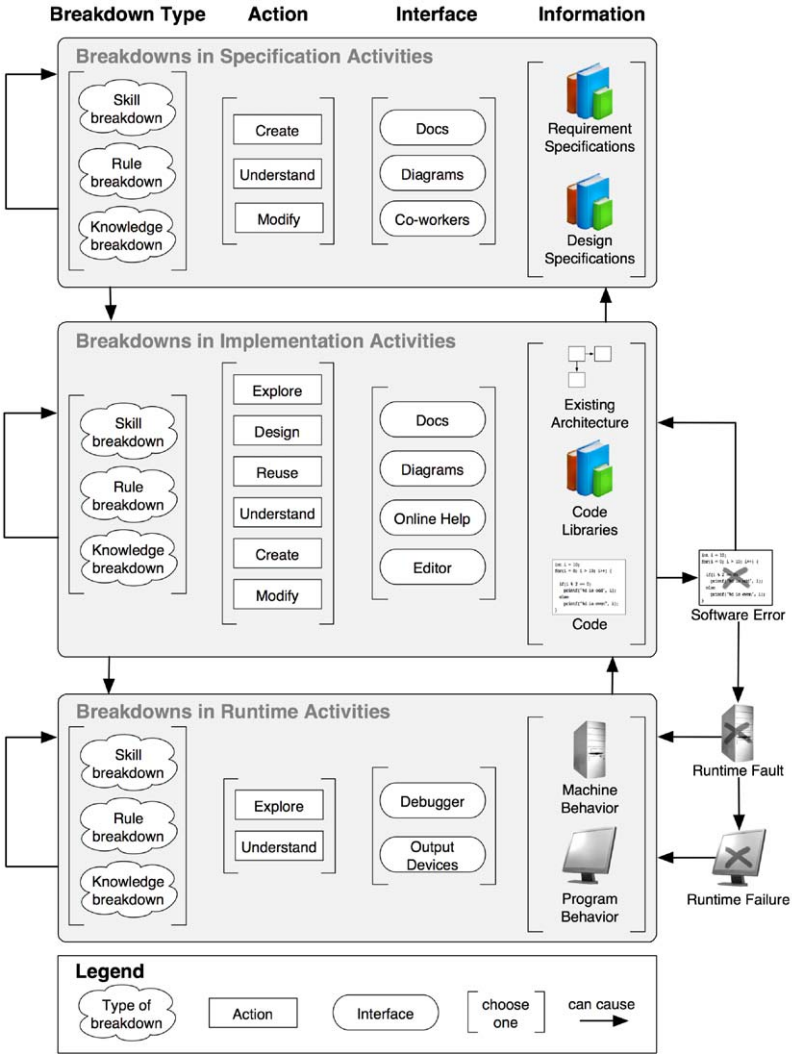

Fig. 4. A framework for describing the causes of software errors based on chains of cognitive breakdowns. Breakdowns occur in specification, implementation, and runtime activities. A single breakdown is read from left to right and consists of one component from each column within an activity. The cause of a single software error can be thought of as a trace through these various types of breakdowns, by following the "can cause" arrows between and within the activities.

three types of actions, one of three types of interfaces, and one of two types of information (therefore, the framework can describe $3 \times 3 \times 3 \times 2 = 54$ types of breakdowns in specification activities). Thus, one possible breakdown described by the framework would be "a knowledge breakdown in understanding a diagram of a

design specification.'' The actions, interfaces, and information for a particular activity are determined by the nature of the activity. For example, in runtime activity, programmers explore and understand machine and program behavior, but they do not create or design it.

Chains of breakdowns are represented by following the arrows in Fig. 4, which denote ''can cause'' relationships. For example, by following the arrow from specification activities to implementation activities, we can say, ''a knowledge breakdown in understanding a diagram of a design specification can cause a knowledge breakdown in modifying code.'' The framework allows all ''can cause'' relationships *within* each activity; for example, during specification, ''a software architect's breakdowns in creating design specification diagrams can cause programmers to have knowledge breakdowns in understanding them.'' The framework also supports relationships *between* activities, as in ''breakdowns in modifying design specification documents can cause breakdowns in modifying code,'' or, ''breakdowns in understanding code in an editor can cause breakdowns in understanding design specification documents.''

In addition to describing ''can cause'' relationships within and between activities, the framework also describes relationships between software errors, runtime faults, runtime failures, and other breakdowns. For example, software errors can cause breakdowns in modifying code before ever causing a runtime fault: when a programmer makes a variable of Boolean instead of integer type, any further code that assumes the variable is of integer type is erroneous. A runtime fault or failure can cause various types of debugging breakdowns if noticed.

While our framework suggests many links between breakdowns, it makes no assumptions about their ordering. High-level software engineering processes, such as the waterfall or extreme programming models, assume a particular sequence of specification, implementation, and debugging activities; models of programming, program comprehension, testing, and debugging assume a particular sequence of programming actions. Our framework describes the causes of software errors in any of these models and processes.

To illustrate how these chains of breakdowns occur, consider the scenario illustrated in Fig. 5. A programmer had little sleep the night before, which causes an *repetition* breakdown in implementing the swap algorithm for a recursive sorting algorithm; this causes a repeated variable reference. At the same time, a *faulty model* knowledge breakdown in understanding the algorithm's specifications causes an *overconfidence* breakdown in implementing a statement in the recursive call; this causes another erroneous variable reference. When he tests his algorithm, the two software errors cause two runtime faults, causing the sort to fail. When observing the failure, the programmer has a *problematic signs* breakdown in observing the program's output because it is displayed amongst other irrelevant debugging output, and he perceives a ''10'' instead of the ''100'' that is on-screen. This causes the programmer to have a *biased reviewing* breakdown in understanding the runtime failure: he forms an incorrect hypothesis about the cause of the failure, and neglects to consider other hypotheses. This invalid hypothesis causes a *selectivity* breakdown in modifying the recursive call, ultimately causing infinite recursion.

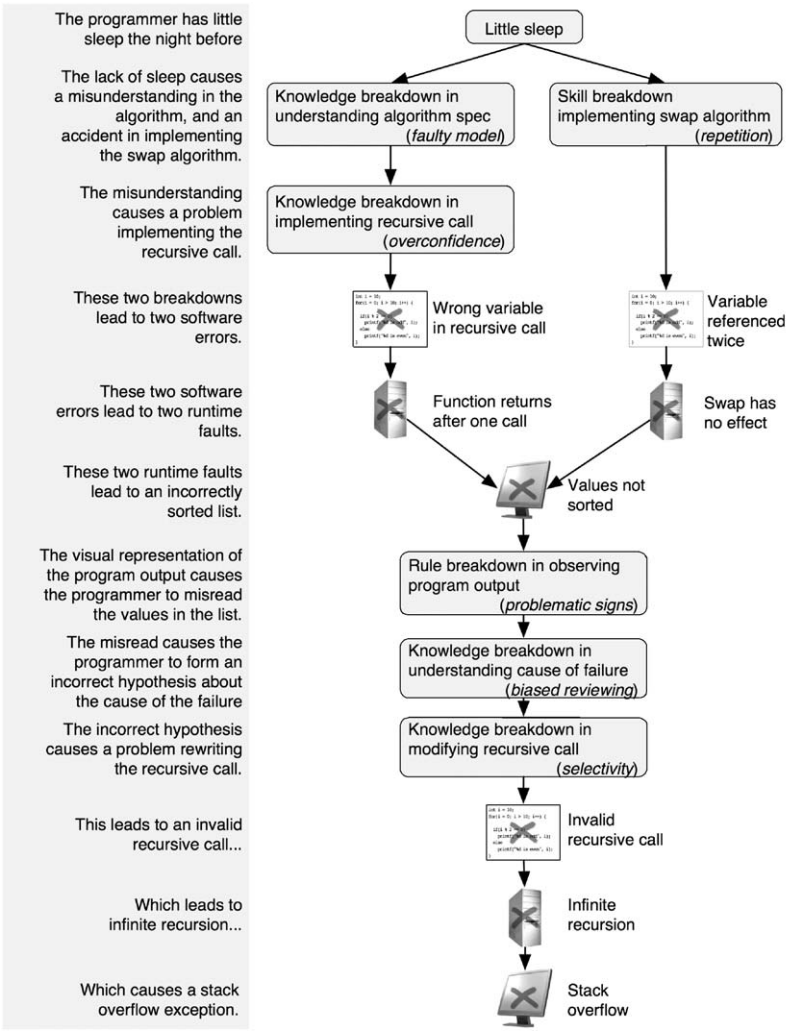

Fig. 5. An example of a chain of cognitive breakdowns, where a programmer has several breakdowns while implementing a recursive sorting algorithm.

4. An empirical methodology for studying the causes of software errors in programming systems

One use of our framework is as a vocabulary for the causes of software errors in a programming system. It enables statements such as "This window in the code editor might make programmers prone to *problematic sign* breakdowns in cutting and pasting code, since it obscures part of the pasted text." In this sense, it can complement other "broad-brush" frameworks such as the Cognitive Dimensions of

Notations [30]. However, we believe the real strength of the framework is in using it to guide the empirical analysis of programming systems, with the goal of collecting design insights and inspirations that would have otherwise not been obvious.

In this section, we describe a methodology for performing such analyses. It has four major steps:

1. Design an appropriate programming task.
2. Observe and record suitably experienced programmers working on the task, using *think-aloud* methodology [31] to capture their decisions and reasoning.
3. Use the recordings to reconstruct chains of cognitive breakdowns by working backwards from programmers' software errors to their causes.
4. Analyze the resulting set of chains of breakdowns for patterns and relationships.

We have used this methodology to study a variety of programming systems in varying contexts, and in Section 5 we discuss study of the Alice programming system [10] as an example. Before doing so, however, we discuss each step of the methodology in detail, highlighting several methodological issues that arise in their execution as well as many recommendations that have come from our experiences.

4.1. Task design

The first and most critical step in our methodology is designing a suitable programming task. Because the task largely determines what is observed, there are several important considerations:

1. *Task complexity*: If the intention of the study is to test the influence of a particular feature of a programming system, will programmers be able to get far enough in the task to reach the interesting observations? For example, if the key focus of the study is on debugging, is the task complex enough to involve debugging, but simple enough that programmers have time to debug?
2. *Uninteresting observations*: Are there any observations that would *not* be interesting? If so, how can one ensure that the task involves as few of these uninteresting interactions as possible?
3. *Strategic variability*: How much will programmers' strategies for the task vary, and is variability desired? Variability helps assess the *range* of a programming system's error-proneness (and is thus important in exploratory studies), but not the *prevalence* of any particular breakdowns.
4. *Level of detail in task specifications*: At what granularity should specifications for the programming task be provided? If the specifications are fairly detailed, defining correctness, and thus identifying runtime failures, will be relatively straightforward. If requirements are provided, but the specifications left unconstrained, then the programmer is ultimately responsible for defining the appropriate behavior of the program. In this case, the programmer is the only one who can deem their program's behavior or code incorrect.

Rather than assuming the answers to any of these questions, we recommend piloting several potential tasks with programmers until any issues are resolved. We describe how we dealt with these issues in our study of Alice [10] in Section 5.

4.2. Observing programmers at work

How should programmers be observed and what should be recorded? Because the underlying assumption of our methodology is that a programming system is prone to a subset of all possible chains of breakdowns described by the framework, recordings should capture all four aspects of a cognitive breakdown. As we described in Section 3, a breakdown consists of four components: the *type* of breakdown, the *action* performed by the programmer, the *interface* used to perform the action, and the *information* acted upon. The latter three components are directly observable. For example, by watching a programmer use a UNIX environment to code a C program, one can observe the programming *interfaces* she uses (emacs, vi, man pages, etc.), the *actions* she performs using these interfaces (editing, shell commands, searching, etc.), and the *information* that she is acting upon (code, makefiles, text console output, etc.).

The only unobservable component of a breakdown is its *type*—one of the many types discussed in Section 2.3.2. Merely analyzing a programmer's actions will not reliably suggest a programmer's goals and decisions, since a single action may have many possible motives. Instead, we suggest using *think-aloud* methodology [31] to elicit the causes of programmers' actions. In think-aloud studies, the experimenter asks participants to provide self-reports of the decision-making and rationale behind their actions.

What constitutes a valid think-aloud study has historically been quite controversial. Ericsson and Simon originally proposed in *Protocol Analysis*: *Verbal Reports as Data* [31] that verbal data is only reliable if it does not require additional attentional resources to verbalize; if it did, data collection would interfere with the task performance. Ericsson and Simon, and others, have since demonstrated that problem-solving tasks that are (1) describable in terms of verbalizable rules and (2) generally without external, time-critical factors, satisfy this condition. Most programming situations seem to satisfy these constraints, although we are unaware of any research verifying this.

There are a number of important guidelines to follow when collecting think-aloud data from programmers. We base our guidelines on Boren and Ramey's recent assessment of think-alouds for usability testing [32]:

- The experimenter should *set the stage*. Participants should understand that *they* are not under study, but rather, the *programming system* is. Furthermore, participants should understand that *they* are the domain experts because they can approach tasks in ways the experimenter cannot. Therefore, while thinking aloud, they are the primary speaker, while the experimenter plays the role of an "interested learner." These roles should be defined explicitly prior to observations and maintained throughout observations.

- The experimenter should take a proactive role in keeping participants verbal reports *undirected*, *undisturbed*, and *constant*. Boren and Ramey recommend using the phrases "Mm hmm" to acknowledge the participant's reports, and "Please continue" or "And now?" as reminders to continue thinking aloud. The experimenter should not ask programmers why they have done something, to avoid biasing participants' explanations, or eliciting fabricated explanations.

Nielson provides a more instructive introduction to think-aloud studies in *Usability Engineering* [5], as well as evidence that the method can be successfully applied by computer scientists with minimal training. For a longer introduction, consult Ericsson and Simon [31].

To actually record breakdowns, we recommend either videotaping programmers at work or recording the contents of their screen using video capture software with an accompanying audio recording. While this may seem like an unnecessarily large amount of data to gather and analyze, anything less than a full recording of a programmer's interaction with a programming system can severely hinder the validity of assessments of the causes of a software error. In our experience, observations such as the pauses between clicks, the code scrolled to, what code is being focused on and even the *speed* of scrolling can all be reliable indicators of a programmer's goals and decisions when combined with verbal utterances. For example, many environments show tool tips when the mouse cursor is hovered over particular code fragments; by only instrumenting a programming system to record high-level actions such as "tool tip shown," "button pressed," and "text deleted," there would be no indication of whether the programmer actually meant to inspect the tool tip, or whether he just happened to leave the mouse cursor at that position while he consulted some printouts on his desk.

4.3. Reconstructing chains of cognitive breakdowns

To reconstruct chains of cognitive breakdowns from a recording, we use a deductive approach in which one asks questions about a software error, runtime fault, runtime failure, or cognitive breakdown in order to determine its cause. This backwards reasoning proceeds until no further causes can be determined from the evidence. We illustrate this process in Fig. 6, which reconstructs the chain presented in Fig. 5.

We begin the reconstruction from the program's runtime failure, a stack overflow exception, by asking the deductive question, "What caused the stack overflow?" Deducing the chain of causality from this failure, to the runtime fault, from the fault to the software error is essentially debugging—to analyze the situation, one must understand the programmer's code well enough to be able to determine all of the software errors that contributed to the program's failure. This deduction can be done objectively, given enough knowledge of the program's code and runtime behavior. We have found that performing this analysis from the videotape often requires repeated rewinding and fast-forwarding, and thus having the video in digital format is quite helpful.

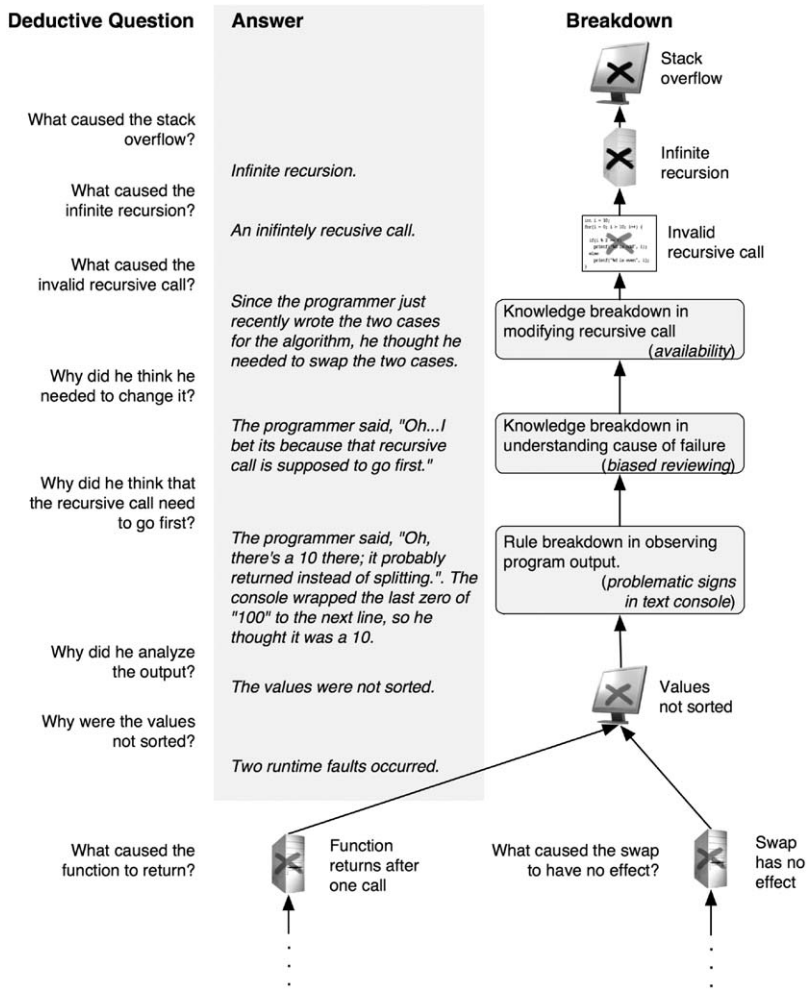

Fig. 6. Deductively reconstructing the causal chain of breakdowns represented in Fig. 5, using a programmer's actions and verbal utterances.

Once the software errors leading to the runtime failure have been deduced, one must determine what types of cognitive breakdowns led to the software error. For example, in Fig. 6 we ask, "What caused the invalid recursive call?" Had the programmer said nothing about his actions, there would have been several explanations, but none with supporting evidence. However, because the programmer said, "Oh, I bet it's because that recursive call was supposed to go first" and then proceeded to move the recursive call in his code, we can be relatively confident that it was an *availability* breakdown: the programmer assumed that his most recent changes to the code were responsible, rather than the swap code. We then proceed to

ask deductive questions about each breakdown, until no further causes can be deduced from the evidence.

In some circumstances, there can be multiple events responsible for a single breakdown, at which point the chain is split in two. For example, in Fig. 6, there are multiple reasons why the sort failed (two runtime faults, two corresponding software errors, and thus at least two cognitive breakdowns). In general, chains can branch at runtime failures (due to multiple runtime faults), at software errors (due to multiple cognitive breakdowns), and at breakdowns (due to multiple external events, such as interface problems or interruptions).

4.3.1. Determining a breakdown's type

How should a breakdown's type be determined? In our analyses, we have been using programmers' verbal utterances and other contextual information to answer deductive questions about some action. For example, if a programmer types the wrong variable name in a method call, our deductive question would be, "Why did the programmer use variable X instead of variable Y?" We answer this question by considering the programmer's past actions and verbal utterances. For example, if the programmer said, "What do we have to send to this method? Um, I think X." we might deduce that he had a *biased reviewing* knowledge breakdown because he was in knowledge-based cognitive activity and only considered one course of action.

To help make these judgments about a breakdown's type, we summarize the various types of skill, rule, and knowledge breakdowns from Section 2.3.2 in Table 6. This table can be used to find an appropriate answer for each deductive question. We have found this table to be an indispensable aid in considering the possible explanations for a programmer's behaviors. In our experience, when considering the context of some action, either a *single* type of breakdown stands out, or none do. If it is unclear which type of breakdown was to blame, the observations are probably insufficient for objectively deducing the cause of the cognitive breakdown. However, even in this case it is useful to record all of the possible breakdowns, since programmers' actions or decisions that have yet to be analyzed may disambiguate a breakdown's type.

Although programmers' verbal utterances can be a valuable and reliable indicator of a breakdown's type, this is only true if the verbal data is analyzed in a reliable way. We recommend testing the reliability of interpretations by having multiple individuals reconstruct a subset of the software errors independently, and then checking for agreement in the types of breakdowns and structure of chains of breakdowns. To check for agreement in types simply involves comparing the categories chosen from Table 6; a more coarse comparison would ensure that analyzers agreed on whether individual breakdowns were *skill*, *rule*, or *knowledge* breakdowns, whereas a finer grained comparison would check for agreement on subtypes of these breakdowns. Checking for agreement on chains of breakdowns involves comparing the causal links between breakdowns; in other words, the answers from the analyzers to each deductive question should be comparable, and each answer should lead to a comparable deductive question.

Table 6
A summary of common types of skill, rule, and knowledge breakdowns, which can be used to answer deductive questions from observations

Detecting skill breakdowns	
Skill-based activity is when…	The programmer… • Is actively executing routine, practiced actions in a familiar context • Is focused internally on problem solving, rather than executing the routine actions
Skill breakdowns happen when…	The programmer… • Is interrupted by an external event (interruption) • Has a delay between an intention and a corresponding routine action (delayed action) • Is performing routine actions in exceptional circumstances (strong habit intrusion) • Is performing multiple, similar plans of routine action (interleaving) • Misses an important change in the environment while performing routine actions (exceptional stimuli) • Attends to routine actions and makes a false assumption about their progress (omission, repetition)
Detecting rule breakdowns	
Rule-based activity is when…	The programmer… • Detects a deviation from the planned-for conditions • Is seeking signs in the environment to determine what to do next
Rule breakdowns happen when…	The programmer… • Takes the wrong action • Misses an important sign (favored signs) • Is inundated with signs (information overload) • Is acting in an exceptional circumstance (favored rules, rigidity) • Misses ambiguous or hidden signs in the environment (problematic signs) • Acts on incomplete knowledge (incomplete knowledge) • Acts on inaccurate knowledge (inaccurate knowledge) • Uses an exceptional, albeit successful rule from past experience as the rule (exception proves rule)
Detecting knowledge breakdowns	
Knowledge-based activity is when…	The programmer… Is executing unpracticed or novel actions • Is comprehending, hypothesizing or otherwise reasoning about a problem using knowledge of the problem space
Knowledge breakdowns happen when…	The programmer… • Makes a decision without considering all courses of action or all hypotheses (biased reviewing) • Has a false hypothesis about something (confirmation bias) • Sees a non-existent relationship between events (simplified causality) • Notices illusory correlation, or does not notice real correlation between events (illusory correlation) • Does not attend to logically important information when making decision (selectivity) • Does not consider logically important information that is unavailable, or difficult to recall (availability) • Is overconfident about the correctness and completeness of their knowledge (overconfidence)

4.4. Analyzing chains of cognitive breakdowns

Once a set of reliable chains of cognitive breakdowns has been reconstructed from observations, there are a wide variety of questions that can be asked:

- What activities are most prone to cognitive breakdowns?
- What aspects of the language and environment are involved in breakdowns?
- What types of actions are most prone to breakdowns?
- How do novice and expert programmers' types of breakdowns compare?
- What breakdowns tend to cause further breakdowns?

It is also important to consider how individual breakdowns are related to the software errors they cause:

- Which types of breakdowns are most prone to cause software errors?
- Which breakdowns due to problems in the programming system are responsible for the most software errors?
- Which kinds of software errors tend to cause further software errors?

We also recommend a higher-level analysis of recurring chains of breakdowns. For example, are there certain patterns of breakdown that almost always lead to software errors? Are there common trends in the contextual factors of these chains that suggest a particular intervention? This type of analysis can be performed by analyzing the set of causal links between breakdowns. For example, for a given data set, how many *availability* breakdowns in understanding runtime failures led to *wrong rule* breakdowns in modifying code? By calculating the frequency of each particular type of causal link between breakdowns, one can get an empirical measure of the relative influence of various usability problems and cognitive failures on the introduction of software errors. We give an example of this type of analysis in the next section.

5. Causes of software errors in Alice

In this section, we present an exploratory study of the causes of software errors in the Alice programming system in order to illustrate the use of our framework and methodology.

5.1. The Alice programming system

The Alice programming system [10] (www.alice.org) is an event- and object-based, concurrent, 3D programming system. Alice is designed to support the development of interactive worlds of 3D objects, and provides many primitive animations such as "move", "rotate" and "move away from." Alice does not support typical object-oriented features such as inheritance and polymorphism. Because it is event-based, Alice provides explicit support for handling keyboard and mouse events, in addition

to conditional events such as "when the world starts" and "while this condition is true."

The Alice 2.0 programming environment, seen in Fig. 7, consists of 5 main views. In the upper left (1) is a list of all of the objects that are in the Alice world, and in the upper middle (2) is a 3D worldview based on a movable camera. The upper right (3) shows a list of global events that the programmer wants to respond to, and the lower left (4) shows the currently selected object's properties, methods, and questions (functions). Lastly, the bottom right (5) is the code view, which shows the method currently being edited. Alice provides a drag-and-drop, structured editing environment in which object's properties and methods are created, modified, and reused by dragging and dropping objects on-screen. This interaction style prevents all syntax and type errors.

When an Alice program is run, the output is displayed in a modal output window, preventing the programmer from interacting with it until the program stops executing. Alice provides no debugger, but properties' values are visible in the property list behind the output window at runtime, and are updated as the program executes. However, because the output window is modal, programmers are restricted to watching the properties that were visible when the program was executed.

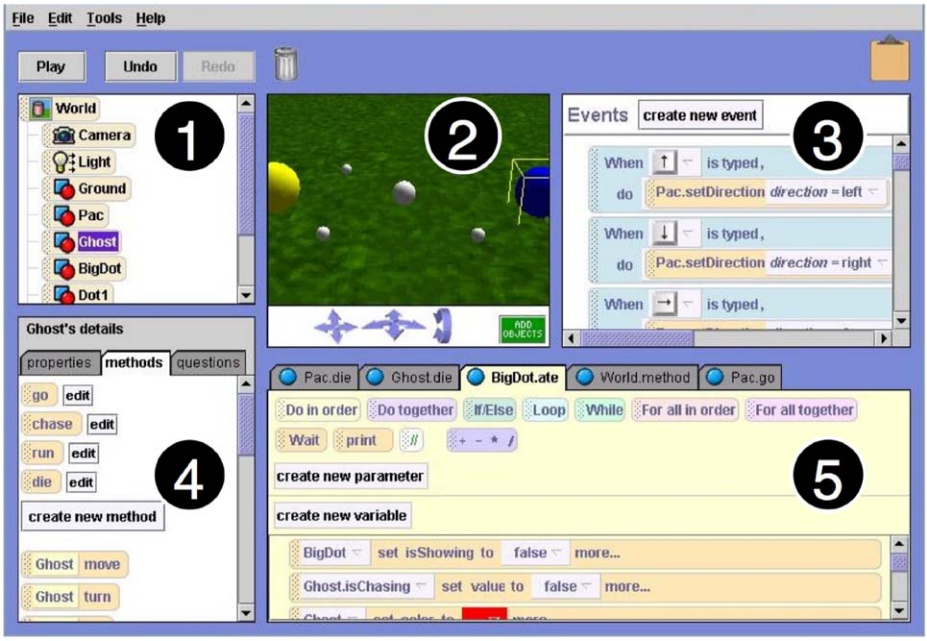

Fig. 7. Alice 2.0, showing: (1) objects in the world, (2) the 3D worldview, (3) events, (4) properties, methods, and questions (functions) for the selected object, and (5) the method being edited.

5.2. The studies

The overall purpose of our studies of Alice was exploratory, and thus we had no explicit hypotheses in mind. We did have two particular goals:

1. Identify common breakdowns in programmers' cognition and consider new types of interfaces that might prevent these breakdowns.
2. Identify common breakdowns due to the Alice environment, and how Alice might be redesigned to prevent these breakdowns.

In designing the studies, we faced the task design issues discussed in Section 4.1:

- *Task complexity*: Because our goals were exploratory, we wanted to observe tasks of varying complexity. Therefore, we decided to observe both experienced Alice programmers working on their own tasks and novice Alice programmers working on a task that we would provide.
- *Uninteresting observations*: Since Alice is a 3D programming system, a large part of writing an Alice program is creating the 3D objects that are manipulated at runtime. We were only interested in Alice as a programming tool, and decided to avoid tasks that required a considerable number of complex 3D objects.
- *Strategic variability*: Because our goals were exploratory, we did not want to assume any particular training with Alice. Therefore, we chose to study both expert Alice programmers who had learned Alice on their own, as well novice Alice programmers, with no particular strategic biases.
- *Level of detail in task specifications*: Again, because our studies were exploratory, we wanted to observe situations where programmers were given specifications, as well as situations where programmers were free to define them as they saw fit.

After iterating on possible tasks and considering various populations in our community, we decided on two studies: one highly unstructured and contextual, and another fairly planned and experimental.

5.2.1. The "Building Virtual Worlds" study

The "Building Virtual Worlds" study (the BVW study) involved 3 Alice programmers (all male engineering undergraduate students) who had been using Alice for six weeks in the "Building Virtual Worlds" course offered at Carnegie Mellon. In this course, programmers were each part of a small team, which typically included a 3D modeler, a sound engineer, and a graphic artist in addition to the programmer. The teams worked together in a computer lab and typically communicated face to face or via e-mail. Each team's class assignment during the time of observations was to use Alice to prototype a complex, interactive, 3D world over the course of two weeks. Since the projects were collaborative and unspecified, the requirements for each programmer's Alice program were in constant flux, and none of the 3 programmer's Alice programs were anything alike.

The experimenter recruited programmers by sending e-mail to each BVW teams' class mailing list and soliciting participation. Three teams expressed interest and the

Table 7
For the BVW study, programmers' self-rated programming language expertise, their total work time, and the tasks that they worked on during observations

ID	Language expertise	Work time (min)	Programming tasks
B1	Average C++, Visual Basic, Java	245	• Parameterize a rabbit's hop animation with speed and height variables • Write code to make tractor beam catch rabbit when in line of sight • Programmatically animate camera moving down stairs • Prevent goat from penetrating ground after falling • Play sound in parallel with character swinging bat.
B2	Above average C++, Java, Perl	110	• Randomly resize and move 20 handlebars in a subway train
B3	Above average C, Java	50	• Import, arrange, and programmatically animate objects involved in camera animation.

experimenter scheduled separate times to observe each at work. When first meeting each team, the experimenter explained that the intentions of the observations were to "learn about how the programmer used Alice." The experimenter described his role as an "interested learner" and described the programmer's role as the primary speaker, making clear that the programmer was the domain expert. The experimenter then requested that the programmer think aloud while working, explaining his decisions and rationale as he worked. Following this briefing, the experimenter began videotaping the computer monitor over the programmer's shoulder using a Digital 8 camcorder while the programmer worked on their own self-initiated tasks using Alice. During observations, the experimenter used the phrases "And now?" and "Please continue" thirty seconds after silence, to remind the programmers to think aloud. If a programmer left the computer to talk to a team member, the experimenter followed the programmer and recorded the discussion. Observations ended when a programmer had to stop working. Programmers were paid $10 per hour for their participation. Table 7 lists programmers' self-rated programming language expertise, their total work time, and the programming tasks that each worked on during observations.

5.2.2. The "Pac-Man" study

The "Pac-Man" study involved 4 novice Alice programmers (all HCI masters students) and was performed individually at a desk with a standard PC and 17" CRT. Programmers were recruited via an e-mail mailing list. In contrast to the BVW study, all 4 programmers were asked to complete the same task, which was to create a simple Pac-Man game with one ghost, four small dots, and one big dot (as seen in Fig. 7). After a 15-min tutorial on how to create code, methods and events,

Table 8
For the Pac-Man study, programmers' self-rated programming language expertise, their total work time, and the requirements for the Pac-Man program that they implemented in Alice

ID	Language expertise	Work time (min)	Programming tasks
P1	Above average Java, C	95	• Pac must always move. His direction should change in response to arrow keys.
P2	Below average C++, Java	90	• Ghost must move in randomly half of the time and towards Pac the other half.
P3	Above average Java, C++	215	• If Ghost is chasing and touches Pac, Pac must flatten and stop moving forever.
P4	Above average Visual Basic	90	• If Pac eats big dot, ghost must run away for 5 seconds, then return to chasing. • If Pac touches running ghost, Ghost must flatten and stop for 5 seconds, then chase again.

programmers were given the same briefing as in BVW study, and were then given the requirements for the Pac-Man game, which are listed in Table 8. Programmers remained at the desk throughout observations, and were videotaped over the shoulder with a Digital 8 camcorder. As with the BVW study, the experimenter used the phrases "And now?" and "Please continue" as reminders to think aloud. Programmers worked for 90 minutes or longer if they wished to work more on the game. Programmers were paid $15 for their participation as long as they completed at least 90 minutes of work. Table 8 lists programmers' self-rated programming language expertise, their total work time, and the requirements for the Pac-Man game that each programmer implemented in Alice. All participants were male except for P2.

5.3. Analyses

Because the programmers in each of the studies were responsible for their own design specifications (the actual implementation of their requirements), we only reconstructed chains based on software errors that caused runtime failures. We did not analyze software errors that did *not* cause runtime failures, because we could not identify them: as discussed in Section 4.1, when design specifications exist only in a programmer's head, the programmer is the only person who can deem that program behavior violates a specification.

The observations resulted in 895 minutes of recordings, all of which was analyzed. In total, it took about 40 h to analyze the 15 hours of recordings. The first phase of analysis was to search each recording for runtime failures by finding incidents where a programmer explicitly labeled some program behavior as incorrect (as in, "What? Pac's not supposed to be bouncing!"). Once these failures were found and timestamps were recorded for each, the next phase was to informally scan the programmers' actions before and after the failure in order to get a sense for what software errors were responsible for the runtime failure; in many cases, the software

errors were obvious because the programmer later found the errors after debugging. Once the software errors were determined, deductive questions were asked about each, and programmers' verbal utterances in close temporal proximity were used to determine the answers. Timestamps were recorded for each of the breakdowns in the chain, along with other contextual details such as the interface and information involved in the breakdown.

One of P2's resulting chains is depicted in Fig. 8. In the figure, the instigating breakdown in creating the specifications for the Boolean logic led to a *wrong action* breakdown in implementing the logic, which led to incorrect logic in the code. At the same time, the P2 had a *problematic sign* breakdown, assuming that a reference to "BigDot" was already included, but off-screen, when in fact it was not. This led to a missing reference error. Both of these software errors caused the conditional to become true after a single dot was eaten, causing Pac-Man to bounce before he had eaten all of the dots. When P2 observed the failure, she had a *biased reviewing* breakdown, only forming one incorrect hypothesis about its cause. This false hypothesis led to a *wrong action* breakdown in modifying the expression, which caused Pac-Man to bounce immediately, before eating *any* dots. However, because P2 had moved the camera position to look down on Pac-Man, the failure was no longer visible (a *problematic signs* breakdown). This caused P2 to have an *illusory correlation* breakdown, where she believed the earlier failure had been repaired because Pac-Man did not *seem* to be bouncing. Twenty minutes later, after repositioning the camera, she noticed that Pac-Man was actually still bouncing at runtime, but experienced an *availability* breakdown, assuming that her recently modified code was to blame, rather than the still incorrect Boolean expression.

One analyzer reconstructed breakdown chains from all of the 7 programmers' runtime failures, and a second analyzer reconstructed chains for a random 10% of the runtime failures, to test for inter-rater reliability. There was approximately 93% agreement on whether breakdowns were knowledge, rule, or skill breakdowns, but less agreement on the sub-types of breakdowns. The causal links between breakdowns in the reconstructed chains were largely the same, although each of the analyzers noticed some breakdowns that the other had not.

Although the time to construct the chains was not recorded, the authors recall that analysis time was largely dependent on how much time the chain of breakdowns covered in the recording: for example, a chain spanned 10 min of video took twice as long to reconstruct than one that spanned 5 min of video. Overall, the analyses spanned approximately a 40 hour week.

5.4. Results

5.4.1. Overall statistics

Over 895 minutes of observations, there were 69 root breakdowns (breakdowns with no identifiable cause) and 159 total breakdowns. These caused 102 software errors, 33 of which led to one or more new software errors. The average chain had 2.3 breakdowns (standard deviation 2.3) and caused 1.5 software errors (standard deviation 1.1). Table 9 shows the proportions of time programmers spent

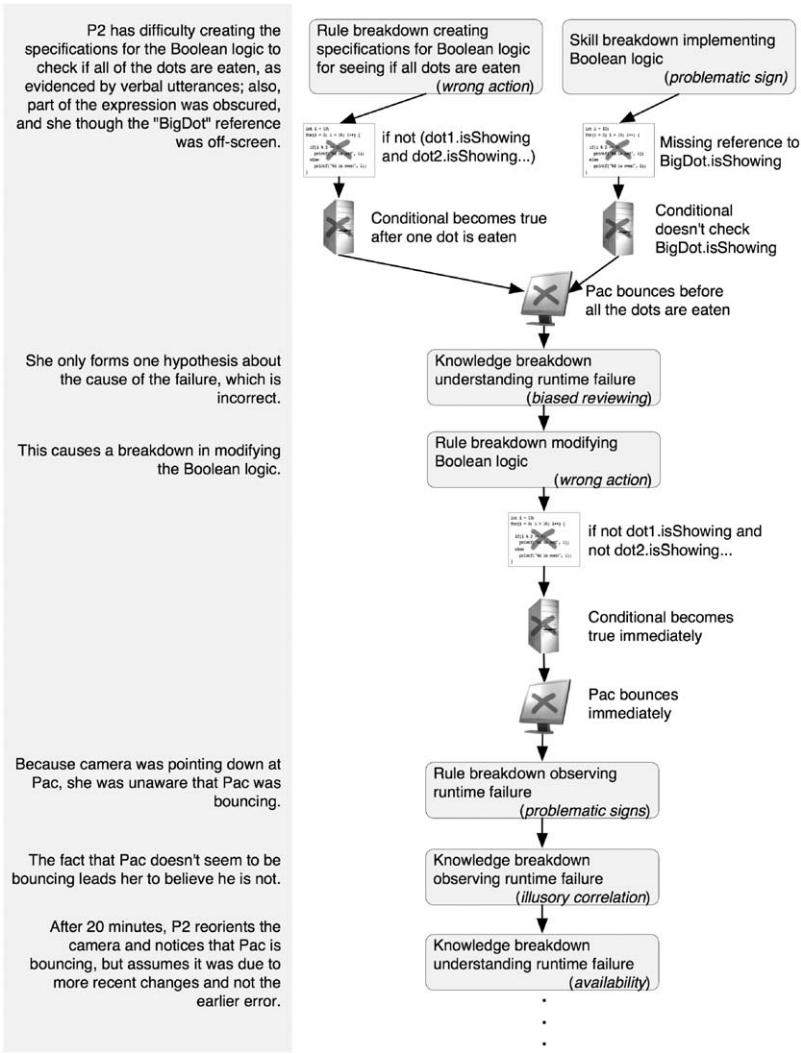

Fig. 8. A segment of one of P2's cognitive breakdown chains. The last breakdown shown here did not cause further breakdowns until 20 minutes later, after the camera position made it apparent that Pac was still jumping.

programming and debugging. On average, 46% of programmers' time was spent debugging (and thus a little more than half was spent implementing code). The BVW programmers, whose code was more complex, had longer chains of breakdowns than the Pac-Man programmers', suggesting that the causes of their software errors were more complex.

As seen in Table 10, about 77% of all breakdowns occurred during implementation activity; these tended to be skill and rule breakdowns in implementing and

Table 9
Programming and debugging time, and the number of software errors, breakdowns, and chains, as well as chain length, by programmer

ID	Programming time (min)	Debugging time (min)		# of software errors	# of breakdowns	# of chains	Average chain length
		Minutes	% of time				Mean (SD)
B1	245	142	58.0%	23	41	10	4.1 (3.5)
B2	110	35	32.8%	16	32	7	4.6 (3.3)
B3	50	11	22.0%	3	5	4	1.2 (0.5)
P1	95	23	36.8%	14	23	11	2.1 (1.7)
P2	90	30	33.3%	7	7	7	1.0 (0.0)
P3	215	165	76.7%	34	44	25	1.8 (1.2)
P4	90	27	30.0%	5	7	5	1.4 (0.5)
Total	895	554	46.4%	102	159	69	2.3 (2.2)

Table 10
Breakdowns split by activity and type

Activity	Type of breakdown	% of all breakdowns
Specification	Skill	0.0%
	Rule	3.1%
	Knowledge	1.2%
	Total	4.4%
Implementation	Skill	22.0%
	Rule	28.3%
	Knowledge	27.0%
	Total	77.4%
Runtime	Skill	8.1%
	Rule	0.0%
	Knowledge	10.1%
	Total	18.2%

modifying artifacts and knowledge breakdowns in understanding and implementing artifacts. About 18% of all breakdowns occurred in runtime activity; these tended to be knowledge or skill problems in understanding runtime failures and faults. The proportion of skill, rule, and knowledge breakdowns were about equal. The root breakdowns of most chains were knowledge breakdowns understanding runtime failures and runtime faults and skill and rule breakdowns implementing code.

Table 11 shows which aspects of Alice were most often involved in cognitive breakdowns. Most breakdowns involved the construction of algorithms and the use of language constructs and animations. This is to be expected, since the majority of

Table 11
Frequency and percent of breakdowns and software errors by type of information and the average debugging time for software errors in each type of information

Type of information	Breakdowns		Software errors		Debugging time
	Frequency	% of all breakdowns	Frequency	% of all errors	Mean (SD) in minutes
Algorithms	37	23.3%	34	33.3%	4.8 (6.2)
Language constructs	35	22.0%	31	30.4%	4.6 (5.5)
Animations	21	13.2%	19	18.6%	7.1 (6.9)
Runtime failures	20	12.6%	—	—	—
Events	18	11.3%	10	9.8%	3.6 (4.2)
Runtime faults	9	5.7%	—	—	—
Data structures	8	5.0%	7	6.9%	3.3 (4.1)
Run-time specification	5	3.1%	—	—	—
Environment	4	2.5%	1	1.0%	1.0 (—)
Requirements	2	1.3%	—	—	—
Software failures	0	0%	—	—	—

the observations were of programmers completely new to the Alice programming system.

Table 12 shows the number of software errors and time spent debugging by problem and action. Most software errors were caused by rule breakdowns in implementing, modifying, and reusing program elements (rather than understanding or observing program elements). The variance in debugging times was high, and the longest debugging times were on rule breakdowns in reusing code and knowledge breakdowns understanding code.

5.4.2. *Significant causes of software errors in Alice*

There were four major causes of software errors in the studies. In each case, the Alice design shared a considerable portion of the blame.

The most common cognitive breakdowns that led to software errors were breakdowns in implementing Alice numerical and Boolean expressions (33% of all breakdowns). Most were *bad rule* breakdowns in implementing complex Boolean expressions. For example, when programmers in the Pac-Man study wanted to test if all of the dots were eaten, their expressions were "*if not* (*dot1.isEaten and dot2.isEaten*…)" which evaluates to true if *any* dots are eaten. This confirms earlier studies by Pane showing that creating Boolean expressions is of considerable difficulty and highly error-prone [33]. In other cases, whether or not they had created a correct expression, programmers suffered from *problematic signs* breakdowns in

Table 12
Software errors and debugging time by cognitive breakdown type and action. Only actions causing software errors are shown

Breakdown	Action	Software errors		Debugging time
		Frequency	% of errors	Mean (SD) in minutes
Skill	Implementing	15	14.7%	5.2 (4.3)
	Modifying	14	13.7%	4.6 (7.1)
	Reusing	4	3.9%	1.2 (1.2)
	Total	23	22.5%	4.0 (5.1)
Knowledge	Implementing	15	14.7%	4.2 (4.8)
	Modifying	5	4.9%	5.4 (4.0)
	Reusing	1	1.0%	5.0 (—)
	Understand	6	5.9%	6.8 (5.7)
	Total	27	26.5%	5.3 (4.2)
Rule	Implementing	23	22.5%	4.2 (3.4)
	Modifying	16	15.7%	4.7 (5.1)
	Reusing	3	2.9%	6.6 (9.3)
	Total	52	51%	5.1 (5.4)

modifying the expressions: for example, after placing operators for *and* and *or* into the code, it was not always obvious which part of the expression they had affected because the change was off-screen or obscured.

With so many software errors introduced because of the implementation breakdowns, the breakdowns in debugging (18% of the total) only complicated matters. These debugging breakdowns were due to knowledge breakdowns in understanding runtime faults and failures. In particular, programmers often generated only a single, incorrect hypothesis about the cause of a failure they observed (*biased reviewing*), and then because of their limited knowledge of causality in the Alice runtime system (*simplified causality*), generated an incorrect hypothesis about the code that caused the runtime fault. Because Alice provides virtually no access to runtime data, there were few ways for programmers to test their hypotheses, except through further modification of their code.

The 18% of knowledge breakdowns in debugging, in turn, were ultimately responsible for nearly all of the 24% of rule and skill breakdowns in modifying code, leading directly to software errors. This was because their hypotheses about the cause of the runtime failure had led them to the wrong code, or led them to make the wrong modification. However, these modification breakdowns were also due to interactive difficulties in modifying expressions. When programmers tried to remove intermediate Boolean operators, they often removed other code unintentionally, and because the structure of the code was not clearly visualized, did not realize they had introduced new software errors during modification.

A final source of software errors, largely independent of the cycles of breakdowns described above, were the reuse breakdowns (7% of the total). These were rule breakdowns in reusing code via copy and paste, caused by *problematic signs* in the copied code. In particular, after pasting copied code into a similar context, programmers began the task of coercing references from the old context to the new context. Oftentimes, several uncoerced properties were off-screen, causing the programmer to overlook the references. These software errors were very difficult to debug, because of *overconfidence* breakdowns in understanding their copied code's correctness. Thus, when programmers attempted to determine the cause of their program's failure, their hypotheses were instead focused on recent changes (*availability* breakdowns). Furthermore, because these software errors caused complex, unpredictable runtime interactions, programmers rarely found them.

We summarize these trends in the model shown in Fig. 9, which portrays the most common causal links between breakdowns in our set of chains. The percentages on each line represent the proportion of each particular type of causal link between breakdowns among all links in our data set; together, they account for approximately 74% of all of the links in our chains (the remaining 26% were each below 2% of our data set, and thus are not shown in the figure).

5.5. Discussion of the studies

Our model of the software errors in Alice is of significant value in understanding the software errors that programmers made in their tasks. We learned that most of the root causes of software errors were from inexperience in creating Boolean expressions and forgetting to fully adapt copied code to a new context, but that the impact of these early software errors was compounded by difficulties with debugging and modifying the erroneous code. In particular, only 18% of the breakdowns that occurred (while forming hypotheses about the causes of runtime faults and failures) were the cause of nearly *all* of the software errors introduced. Therefore, even in our simple tasks, there were complex relationships between the programming system's interfaces, the programmers' cognition, and the resulting software errors.

In addition to providing this high-level view of the common breakdowns in using Alice, the *process* of reconstructing the chains of breakdowns directly inspired several design ideas for error-preventing programming tools. For example, the dozens of *biased reviewing* breakdowns in understanding runtime failures provided a rich source of inspiration for the design of the *Whyline*, a new question-based debugging interface [34]. In our observations, immediately after programmers saw their program fail, they asked a question of one of two forms: *why did* questions, which assumed the occurrence of an *unexpected* runtime action, and *why didn't* questions, which assumed the absence of an *expected* runtime action. It was immediately obvious from looking at these chains that the programmers' implicit assumptions about what did or did not happen at runtime had gone unchecked, which led to a lengthy and error-prone debugging session to determine if their false hypothesis was correct. The fundamental idea behind the Whyline—that debugging

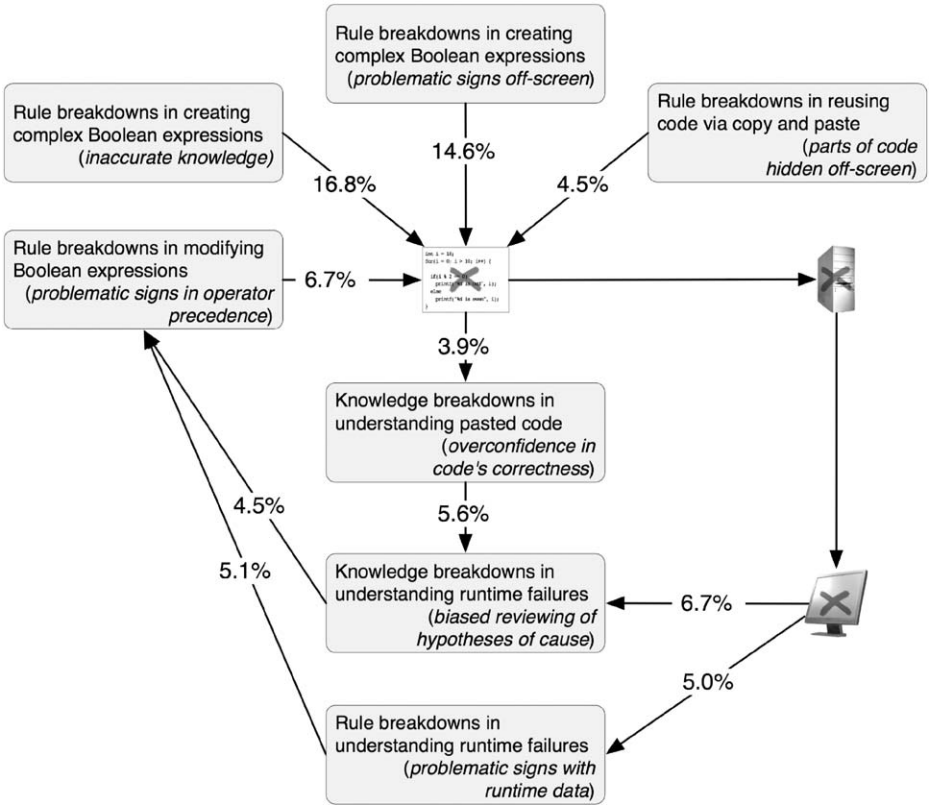

Fig. 9. A model of the major causes of software errors in Alice during programmers' tasks. Each line represents a particular type of causal link between one type of breakdown and another, where the number on the line represents the proportion of the particular type of link out of all links in all chains. Note that we do not include numbers for the links between software errors, runtime faults, and runtime failures, since we only recorded software errors that led to runtime failures.

tools should directly help programmers explore valid hypotheses about the causes of a runtime failure before they consider a false hypothesis—comes directly from the use of our methodology.

Not only did the data from our studies provide the key inspiration for the Whyline, but it also provided a valuable source of information for specific design decisions. For example, since we observed that programmers could easily verbalize their why questions during the think-aloud, why not directly support an interface for asking these questions? Therefore, the Whyline allows programmers to ask questions explicitly about their program's failure by click a "Why" button, which reveals a hierarchical menu of questions to ask about the program's execution. By allowing programmers to choose from a set of *why did* and *why didn't* questions rather than

generate the question themselves, programmers are prevented from forming incorrect hypotheses about their program's runtime behavior altogether.

We also used our observations to determine the space of possible answers to programmers' questions, finding three types:

1. *False propositions*: The programmer's assumption is false. The answer to "Why didn't this button's action happen?" may be that it did, but had no visible effect.
2. *Invariants*: The runtime action always happens (*why did*), or can never happen (*why didn't*). The answer to our button question may be that an event handler was not attached to an event, so it could never happen.
3. *Data and control flow*: A chain of runtime actions led to the program's output. For example, a conditional expression, which was supposed to fire the button's action, evaluated to false instead of true.

For example, had the Whyline been available when the programmer in Fig. 8 noticed that Pac was jumping, she could have pressed the "Why" button and asked, "Why did Pac move up?" The Whyline would have shown the runtime actions directly relevant to her question: the execution of the Boolean expression, the animation moving Pac-Man up, and so on. This way, any implicit assumptions about what did or did not happen at runtime could have been explicitly addressed in the answer.

We have since performed evaluations of the Whyline's effectiveness [34] by comparing identical debugging scenarios done with and without the Whyline, and found that the Whyline reduced debugging time by nearly a factor of 8, enabling programmers to complete 40% more tasks than without the Whyline. These dramatic improvements are the direct result of using our framework and methodology to uncover non-obvious causes of software errors in Alice.

6. Discussion

Based on our experiences in using our framework and methodology, we believe they support many important aspects of programming system design, including reasoning about and studying software errors, as well as inspiring the design of new error-preventing programming tools.

6.1. Using the framework to reason about software errors

As we have seen, prior research on software errors is somewhat fragmented and inconsistent. Classifications have not clearly separated software errors from their causes or their manifestations in program behavior. Our framework provides a consistent and well-defined vocabulary for talking about software errors and their causes. In this way, it can be used as a companion to similar frameworks, such as Green's Cognitive Dimensions of Notations [6]. Cognitive Dimensions have been used to analyze the usability of many visual and professional programming

languages [7,8,35], but none have addressed the causes of software errors. Future studies could identify relationships between dimensions of notations and the causes of software errors. For example, consider *viscosity*, the resistance to local changes. What types of cognitive breakdowns is a viscous interface prone to? Another dimension is *premature commitment*, or the requirement that a user makes some decision before important information is available. In design activities, what types of breakdowns is an interface requiring premature commitment properties prone to? Answering such questions may create a valuable link between salient interactive dimensions of programming systems and their error-proneness.

Our framework also clearly identifies approaches to preventing software errors. For example, software engineering can focus on preventing breakdowns when understanding, creating and modifying specifications. Computer science education can focus on helping programmers prevent knowledge and rule breakdowns, by exposing students to many types of programming, testing, and debugging activities. Programming systems can focus on preventing the rest of the breakdowns that software engineering and education cannot prevent, through less error-prone languages, better support for testing and debugging, and with improved program comprehension and visualization tools.

6.2. Using the methodology to study software errors

In addition to helping reason about software errors, our framework has a number of implications for the empirical study of software errors and programming activity in general. For example, while von Mayrhauser and Vans' Integrated Comprehension Model [24] provides a sophisticated understanding of programmer's various types of mental models, it lacks any mention of problems in forming mental models of specifications or program's static or dynamic semantics. Identifying areas where specification breakdowns can occur may help future studies of program comprehension explicitly link aspects of the comprehension process to specific types of breakdowns. Our model also informs models of debugging, such as Gilmore's [27]. He argues that programmers compare mental representations of the problem and program, but does not account for breakdowns in knowledge formation or mismatch correction, which likely affects debugging in predictable ways.

Because the framework is descriptive, it also supports the objective comparison of software errors within and between programming environments, programs, languages, tasks, expertise, and other factors. Future studies can perform summative comparisons of different programming systems' abilities to prevent breakdowns, which would allow statements such as "language A is more prone to knowledge breakdowns in reusing standard libraries than language B." This is in contrast to existing methodologies, such as Cognitive Dimensions, Cognitive Walkthroughs, and Heuristic Evaluation, which all produce fairly subjective results which cannot be used for comparisons.

Although we have no experience using the framework and methodology for studying "programming in the large," we suspect that our techniques could be easily applied in less-controlled settings. In fact, our study of students in "Building Virtual

Worlds'' was very similar to industry settings, just at a smaller scale: programmers were constantly interrupted, specifications were constantly changing, and each programmer's attention was continuously divided among programming and numerous non-programming tasks. Despite these circumstances, there was little difficulty in collecting the necessary data with a video camera, nor was there difficulty analyzing the data. This is in stark contrast to controlled experiments, which require considerable changes to the way people normally work in order to obtain reliable results.

Despite the several potential contributions our methodology may have for the empirical study of software errors, the methodology itself does have several potential limitations. First, although other researchers have already expressed interest in using the methodology, we have no evidence that it is easily learnable by programming system designers. Because it involves human subjects, it necessarily requires more experience with empirical studies than less formal techniques such as Cognitive Walkthroughs [4] and Cognitive Dimensions [30]. Similarly, another limitation may be the methodology's practicality. As we stated in Section 5.3, it took about 40 hours to analyze about 15 hours of observations across 7 programmers. While this is considerably more time than so-called ''discount'' usability methods take, in our experience, our methodology has resulted in far more concrete and actionable insights.

Another potential limitation is our methodology's replicability. Although we have successfully used the methodology on a wide variety of systems, including Alice, Eclipse, and Visual Studio.NET, we only have a small amount of evidence that other researchers would generate the same data under similar study conditions. To complicate matters, we have found that even the smallest interactive details of a programming system can cause cognitive breakdowns, and thus with programming systems in constant flux, replicability might be unproductive. To some extent, these are only limitations if the data *needs* to be replicable. For example, if the goal of the study is to verify some hypothesis about software errors, then replicability is the chief interest: a general hypothesis requires generalizable data. On the other hand, if the goal of the study is simply to inspire new tools and language designs, *representativeness* is far more important.

One final limitation is the issue of ''fluid requirements''—requirements that are constantly being redefined either because they are unstated and up to the programmer or because they are defined by some other party. Because we define software errors relative to design specifications, it is only feasible to reconstruct chains of breakdowns when there is some indication of a program's requirements. Otherwise, it is difficult to know what constitutes a software error. In the studies described in this paper, we dealt with this issue by limiting our analyses to program behaviors that the programmers themselves said were runtime failures. It is unclear how this limitation has influenced the representativeness of our data.

6.3. Using the framework and methodology for design

Design is never straightforward, and the design of complex error-preventing programming tools is no exception. Nevertheless, we believe our framework and methodology can support the design of such tools in several ways.

6.3.1. Design guidance

One use of our framework is as guidance; for example, by expressing it in terms of heuristics, we can help focus programming system designers on the important issues. We list ten heuristics in Table 13, which we derive from the summary of cognitive breakdowns in Table 6. While most of these heuristics are not specific to programming, few of them have been taken into account in the design of programming systems. For example, heuristic 2 is important to keep in mind in several programming-related fields. When designing documentation standards, software architects should highlight exceptions in software's behavior to prevent programmers from making assumptions about other aspects of the system's behavior. Language designers should weigh the convenience of operator overloading against its cost: programmers unfamiliar with the particular semantics of an overloaded operator may make erroneous assumptions about the program's runtime semantics. Heuristic 2 also suggests that designers of future versions of UML notation should consider notations for identifying exceptional behaviors that software engineers would otherwise assume they understood. Designers of testing and debugging tools might consider identifying uncommon runtime circumstances to programmer's attention.

6.3.2. Design evaluation

Our framework and methodology can also support formative design by acting as an early evaluative tool. For example, there is no requirement that chains of breakdowns must be reconstructed from interactions with a *functional* prototype. We have successfully used our methodology on Wizard of Oz and paper prototypes of the Whyline, in order to determine if it would help programmers to only make correct assumptions. The only requirement is that the system behavior is specified, and that the experimenter reliably follows these specifications. These types of formative studies can be quite valuable in making design decisions before fully implementing a system.

Table 13
Ten heuristics for designing error-preventing programming systems

Heuristics for preventing cognitive breakdowns in programming
1. Help programmers recover from interruptions or delays by reminding them of their previous actions
2. Highlight exceptional circumstances to help programmers adapt their routine strategies
3. Help programmers manage multiple tasks and detect interleaved actions
4. Design task-relevant information to be visible and unambiguous
5. Avoid inundating programmers with information
6. Help programmers consider all relevant hypotheses, to avoid the formation of invalid hypotheses
7. Help programmers identify and understand causal relationships, to avoid invalid knowledge
8. Help programmers identify correlation and recognize illusory correlation
9. Highlight logically important information to combat availability and selectivity heuristics
10. Prevent programmer's overconfidence in their knowledge by testing their assumptions

6.3.3. Design inspiration

Another way our framework and methodology can support design is by inspiring new design ideas. We gave a specific example of this in Section 5.5, in discussing the inspiration for the design of the Whyline. In general, we believe our framework and methodology have several important advantages compared to more traditional methods of finding design inspiration such as brainstorming and interviews:

- Traditional design methods are often heavily biased by designers' experience and intuition, and focus less on observing actual work. Our framework and methodology provide a way of performing structured observations of *actual* programmers using *actual* programming systems.
- Our methodology results in data that can be reanalyzed. We have frequently gone back to video recordings to answer questions that came up in prototyping and changed our design as a result. This is often impossible with interviews and brainstorming, since they do not record actual interactions.
- Because our methodology captures the actual context of programmers' breakdowns, it is much easier to imagine how a new kind of tool might be used and how it might prevent software errors. Traditional design methods often require the programmer, the task, and all other context to be fabricated.

7. Conclusions

Experiences using our framework and methodology to study Alice have been quite positive. Our observations led directly to design inspirations for highly successful debugging tool, as well as several other ideas for preventing software errors in programming systems, which are currently in development. The methodology is practical and relatively low cost, given the number of design ideas our data have given rise to. We encourage other researchers to apply our framework and methodology to further programming systems and share their experiences.

Acknowledgements

We would like to thank the programmers in our studies for their participation and the faculty and students of the Human-Computer Interaction Institute for their constructive comments and criticisms of this work. This work was partially supported under NSF grant IIS-0329090 and by the EUSES Consortium via NSF Grant ITR-0325273. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF.

References

[1] J. Reason, Managing the Risks of Organizational Accidents, Ashgate, Aldershot, 1997.

[2] G. Tassey, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology RTI Project Number 7007.011, 2002.

[3] B.E. John, D.E. Kieras, Using GOMS for user interface design and evaluation: which technique?, ACM Transactions on Computer-Human Interaction 3 (4) (1996) 287–319.
[4] C. Wharton, J. Rieman, C. Lewis, P. Polson, The cognitive walkthrough: a practitioner's guide, in: J. Nielsen, R.L. Mack (Eds.), Usability Inspection Methods, Wiley, New York, 1994.
[5] J. Nielsen, Usability Engineering, Academic Press, Boston, 1993.
[6] T.R.G. Green, Cognitive dimensions of notations, in: A. Sutcliffe, L. Macaulay (Eds.), People and Computers V, Cambridge University Press, Cambridge, UK, 1989, pp. 443–460.
[7] T.R.G. Green, M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, Journal of Visual Languages and Computing 7 (1996) 131–174.
[8] S. Clarke, Evaluating a new programming language, 13th Workshop of the Psychology of Programming Interest Group, Bournemouth, UK, 2001, pp. 275–289.
[9] A. Blackwell, First steps in programming: a rationale for attention investment models, IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, 2002, pp. 2–10.
[10] M. Conway, S. Audia, T. Burnette, D. Cosgrove, K. Christiansen, R. Deline, J. Durbin, R. Gossweiler, S. Koga, C. Long, B. Mallory, S. Miale, K. Monkaitis, J. Patten, J. Pierce, J. Shochet, D. Staack, B. Stearns, R. Stoakley, C. Sturgill, J. Viega, J. White, G. Williams, R. Pausch, Alice: lessons learned from building a 3D system for novices, Proceedings of CHI 2000, The Hague, The Netherlands, 2000, pp. 486–493.
[11] J.R. Anderson, R. Jeffries, Novice LISP errors: undetected losses of information from working memory, Human-Computer Interaction 1 (2) (1985) 107–131.
[12] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, S. Yang, Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm, Journal of Functional Programming 11 (2) (2001) 155–206.
[13] R. Panko, What we know about spreadsheet errors, Journal of End User Computing 2 (1998) 15–21.
[14] J.D. Gould, Some psychological evidence on how people debug computer programs, International Journal of Man-Machine Studies 7 (2) (1975) 151–182.
[15] M. Eisenberg, H.A. Peelle, APL learning bugs, APL Conference, Washington, DC, 1983, pp. 11–16.
[16] W.L. Johnson, E. Soloway, B. Cutler, S. Draper, Bug catalogue: I, Yale University, Boston, MA, Technical Report 286, 1983.
[17] D.N. Perkins, F. Martin, Fragile knowledge and neglected strategies in novice programmers, empirical studies of programmers, 1st Workshop, Washington, DC, 1986, pp. 213–229.
[18] D. Knuth, The errors of TeX, Software: Practice and Experience 19 (7) (1989) 607–685.
[19] M. Eisenstadt, Tales of debugging from the front lines, Empirical Studies of Programmers, 5th Workshop, Palo Alto, CA, 1993, pp. 86–112.
[20] J. Reason, Human Error, Cambridge University Press, Cambridge, UK, 1990.
[21] J.G. Spohrer, E. Soloway, Analyzing the high frequency bugs in novice programs, Empirical Studies of Programmers, 1st Workshop, Washington, DC, 1986, pp. 230–251.
[22] S.P. Davies, Knowledge restructuring and the acquisition of programming expertise, International Journal of Human-Computer Studies 40 (4) (1994) 703–726.
[23] R.L. Shackelford, A.N. Badre, Why can't smart students solve simple programming problems?, International Journal of Man-Machine Studies (38) (1993) 985–997.
[24] A.v. Mayrhauser, A.M. Vans, Program understanding behavior during debugging of large scale software, empirical studies of programmers, 7th Workshop, Alexandria, VA, 1997, pp. 157–179.
[25] H. Simon, Rational choice and the structure of the environment, Psychological Review 63 (1956) 129–138.
[26] I. Vessey, Toward a theory of computer program bugs: an empirical test, International Journal of Man-Machine Studies 30 (1) (1989) 23–46.
[27] D.J. Gilmore, Models of debugging, Acta Psychologica (78) (1992) 151–173.
[28] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, C. Cook, Does continuous visual feedback aid debugging in direct-manipulation programming systems? Conference on Human Factors in Computing, 1997, pp. 22–27.

[29] C.L. Corritore, S. Wiedenbeck, Mental representations of expert procedural and object-oriented programmers in a software maintenance task, International Journal of Human-Computer Studies 50 (1) (1999) 61–83.
[30] A. Blackwell, T. Green, Notational systems—the cognitive dimensions of notations framework, in: J.M. Carroll (Ed.), HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science, Morgan Kaufmann, San Francisco, CA, 2003.
[31] K.A. Ericsson, H.A. Simon, Protocol Analysis: Verbal Reports as Data, MIT Press, Cambridge, MA, 1984.
[32] M.T. Boren, J. Ramey, Thinking aloud: reconciling theory and practice, IEEE Transactions on Professional Communication 43 (3) (2000) 261–278.
[33] J.F. Pane, B.A. Myers, Tabular and textual methods for selecting objects from a group, Proceedings of VL 2000: IEEE International Symposium on Visual Languages, Seattle, WA, 2000, pp. 157–164.
[34] A.J. Ko, B.A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, CHI 2004, Vienna, Austria, 2004, pp. 151–158.
[35] F. Modugno, T.R.G. Green, B.A. Myers, Visual programming in a visual domain: a case study of cognitive dimension, Proceedings of Human-Computer Interaction '94, People and Computers IX, Glasgow, Scotland, 1994, pp. 91–108.