A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems

Andrew J. Ko and Brad A. Myers

Human-Computer Interaction Institute

School of Computer Science

Carnegie Mellon University

Send correspondence to:

Andrew J. Ko

Human-Computer Interaction Institute

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213 USA

E-mail: ajko@cmu.edu

Fax: 412-268-1266

Phone: 412-401-0042

Abstract

Programmers' work is often defined by the correctness, robustness, and flexibility of their code, rather than the efficiency with which they produce it. This makes current usability analysis techniques ill-suited to analyze programming systems, since their focus is more on problems with learnability and efficiency of use, and less on error-proneness. We propose a framework and empirical methodology that focuses specifically on describing the causes of software errors in terms of chains of cognitive breakdowns. The framework is derived from past classifications of software errors, psychological studies of software errors, and research on the mechanisms of human error. Our experiences using the framework to study errors in the Alice programming system suggests that our methodology can help highlight common causes of software errors, and provide important design knowledge for reducing programming systems' error-proneness. We believe our contribution has important implications for programming system design, software engineering, the psychology of programming, and computer science education.

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% to 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 errors has also increased. Because this trend shows no sign of slowing, there is considerable interest in designing programming systems that can demonstrably prevent software errors, and better help programmers diagnose and repair the unprevented errors.

Unfortunately, the design and evaluation of such "error-robust" programming systems still poses a significant of a 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 the bottlenecks are more often in repairing errors than in learning. Even with the more design-oriented techniques,

understanding a programming system's error-proneness has been something of a descriptive dilemma. Nieslen's Heuristic Evaluation suggests little more than to prevent user errors by finding common error situations [5]. Green's Cognitive Dimensions [6], though successfully 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 the causes of software errors in a programming system. Our approach is to integrate three strands of research:

- Past classifications of software errors;
- Models and theories of the cognitive causes of software errors; and
- Research on the cognitive mechanisms of human error.

From this prior work, we derive a framework for describing *chains of cognitive breakdowns* that lead to software errors, 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 visual languages and environments, but also for guiding the design of new error-robust languages, environments, and visualizations.

This paper is divided into six parts. In the next section, we review past classifications of software errors, models and theories that suggest the causes of software errors, and research on human errors in general. In Section 3, we describe our framework and in Section 4 we detail 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 causes of software errors in the Alice

event-based 3D programming system [10]. We end in Section 6 with a discussion of the strengths and applicability of the framework to programming system design, software engineering, psychology of programming, and computer science education.

1 Software Errors: Definitions, Classifications and Causes

In this section, we review classifications of software errors, empirical studies, and cognitive research on human error. We do this with the intent of integrating the causes and characteristics of software errors into a single framework. To help frame our discussion, let us first clarify some relevant terminology.

1.1 Software Errors and Related Terminology

The goal of software engineering is to build a product that meets a particular need. In the software development process, the *correctness* of a software system can be defined relative to this need at many levels of abstraction. These include:

- *Users' expectations* of the software's behavior and functionality;
- A software designer's interpretation of users' 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.

Thus, when we use the term *software error*, we are speaking relative to a certain specification of a system's behavior. Because we are interested in what a programming system can do to prevent software errors, we will consider errors relative only to *design*

specifications. While the causes of software errors certainly include problems with the requirement and design specifications, these problems are typically outside the control of programming systems (and thus outside the scope of this paper).

Relative to design specifications, we define three terms: *software errors*, *runtime faults* and *runtime failures*. 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 design specifications. Of course, a program's behavioral requirements may also be in terms performance, usability, and security, among other software quality attributes. Runtime failures are usually the first indication to the programmer that their program contains errors. A *runtime fault* is a machine state that may cause a runtime failure. For example, a runtime fault may be 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 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 one or more software errors exist, software errors do not *always* cause runtime faults, and runtime faults do not *always* cause runtime failures. The relationships between these three concepts are illustrated in Figure 1.

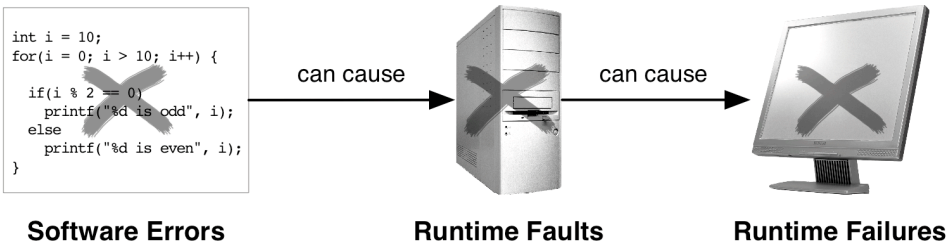

Figure 1. The relationship between software errors in code, runtime faults during execution, and runtime failures in program behavior. These images will be used to represent these three concepts throughout this paper.

Using these definitions, a number of other terms can be clarified. A *bug* is an amalgam of one or more of 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 repairing them. *Testing* involves searching for runtime failures and recording information about runtime faults to aid in debugging. *Corrective* maintenance aims to discover and repair (test and debug) software errors, whereas *adaptive* maintenance aims to change the requirements and code accordingly. Thus, adaptive maintenance can introduce software errors by merely changing specifications.

The last term we define is *programming system*. A programming system consists of three components: (1) *interfaces*, which constitute a programming environment; (2) *information*, including program code and runtime data, which the programmer creates, manipulates, searches, and reveals via the programming environment's interfaces; and (3) *notations*, which are the formalisms in which information is represented. These three

components and their relationships are illustrated in the columns of Figure 2. For example, a programmer interacts with code, which is represented in a particular language notation, via the editing environment. A programmer interacts with the machine's behavior, which is in terms of memory, registers, instructions, call stacks, and so on, using a debugger.

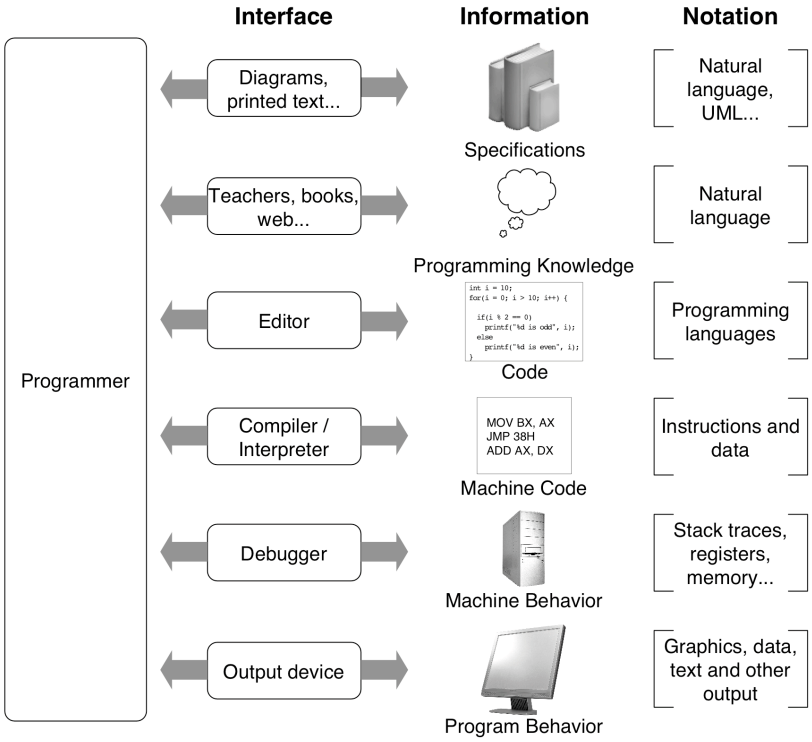

Figure 2. A programming system, which includes all of the interfaces, languages, and other tools that a programmer uses to accomplish programming tasks.

1.2 Classifications of Software Errors

In the past three decades, there has been little work in classifying and describing software errors. Yet, the work that has been done was largely successful in motivating

many novel and effective tools to help programmers find and repair software errors. For example, in the early ‘80’s, the Lisp Tutor drew heavily from analyses of novices’ 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 summarizes some of the most often cited classifications chronologically. In hindsight, it is clear that these classifications are not about software errors alone. Rather, they combine many aspects of software errors, runtime faults, and runtime failures.

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

Study	Bug / Error / Cause	Description	Author's Comments
Gould *1975* [14] *Novice Fortran*	Assignment bug	Errors assigning variables values	Requires understanding of behavior
	Iteration bug	Errors iterating	Requires understanding of language
	Array bug	Errors accessing data in arrays	Requires understanding of language
Eisenberg 1983 [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. *1983* [16] *Novice Pascal*	Missing	Omitting required program element	Errors have contexts: input/output, declaration, initialization and update of variables, conditionals, scope delimiters, or combinations.
	Spurious	Unnecessary program element	
	Misplaced	Required program element in wrong place	
	Malformed	Incorrect program element in right place	
Sphorer & Soloway *1986* [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	
	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 *1989* [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	Errors using and changing 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 1993 [19] *Industry experts COBOL, Pascal, Fortran, C*	Clobbered memory	Overwriting memory, subscript out of bounds	Also identified why 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	
Panko 1998 [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	

We have analyzed these classifications and have identified four salient aspects of software errors. The first is the *surface quality* of a software error. This refers to the particular syntax, language construct, data structure, code library, or other programming information involved in a software error. 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. Clearly, these categories are greatly influenced by the particular language being used to write a program.

A second aspect of software errors is the *cognitive issue* that likely caused the software error. The classifications mention knowledge issues, such as a programmer's lack of knowledge about language syntax, control constructs, data types, and other programming concepts. Eisenberg's *inventive bug*, Sphorer and Soloway's *data-type inconsistency*, and Johnson's *misplaced* and *malformed* categories all refer to knowledge issues. There are also attentional issues, referring to software errors that were likely due to forgetting or a lack of vigilance. Knuth's *forgotten function* category and Eisenstadt's *variable bugs* are good examples. There are also strategic issues, referring to problems like unforeseen code interactions or poorly designed algorithms. Eisenstadt's *design logic bugs* and Knuth's *surprise scenario* category are good examples.

A third aspect of software errors is the *programming activity* in which the cause of the software error occurred. For example, some categories blame specification activity, in which the programmer's invalid or inadequate comprehension of the design specifications led to error. Knuth's *mismatch between modules* bugs and Sphorer and Soloway's *expectation and interpretation* problems are good examples. Another activity is algorithm design, in which the problematic design of an algorithm or unforeseen

interactions with other code led to error. Sphorer and Soloway's *plan dependency problem* is a one example. There is also mention of testing and debugging activity, in which a software error caused a common type of runtime fault. Eisenstadt's *clobbered memory* bugs and Sphorer and Soloway's *boundary* problems are good examples.

A fourth and final aspect of software errors is the type of *action* that led to error. The actions mentioned in the classifications are similar to those in Green's Cognitive Dimensions of Information Artifacts [6]. We list all six actions in Table 2 with examples of each action in programming activity. One action is *creation*: programmers can introduce software errors when *creating* code, but also, the creation of specifications can predispose software errors (as in Sphorer and Soloway's *expectation/interpretation* problems). Other actions include *modifying* specifications and code, *reusing* existing code, *designing* software architectures, algorithms, and objects, and *exploring* code and runtime data. Finally, programmers spend a considerable amount of time *understanding* specifications, data structures, languages, and other information.

Table 2. Actions performed during programming activity, adapted from Green's Cognitive Dimensions of Information Artifacts [6].

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.

Although these classifications sometimes confuse software errors, runtime faults, and runtime failures, we can learn many important things from them. For example, a particular software error has many possible causes, including cognitive problems with knowledge, attention, and strategies. What looks like an erroneously coded algorithm on

the surface may have been caused by an invalid understanding of the specifications, a lack of expertise, inattention, or some other cause. Because of this, we can see that understanding software errors solely by their surface qualities paints an incomplete picture. We also learn that these cognitive problems can occur in many activities: during the creation and comprehension of specifications, during algorithm and module design, while writing a variable name and even while testing and debugging. Finally, we learned that there are many types of programming actions that can lead to error.

1.3 Human Error in Programming Activity

Classifications of software errors have given us a general sense of how cognitive problems are manifested into software errors. However, to understand how the interaction between a programmer and a programming system can lead to software errors, we need to discuss the underlying cognitive mechanisms of human error. James Reason, through his work in *Human Error* [20], provides a solid foundation for understanding these mechanisms. In this section, we adapt two aspects of his research to the domain of programming: (1) a high-level understanding of the causes of failure, and (2) a classification of three general failures of human cognition.

1.3.1 High-level 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. 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." Thus, the fundamental idea is that

there are several layers to complex systems, and each layer may have a number of latent errors that predispose failure. Layers also have a set of defenses, which prohibit latent errors from becoming active errors. When we combine these ideas, we see that failures are ultimately due to a causal chain of failures both within and between layers.

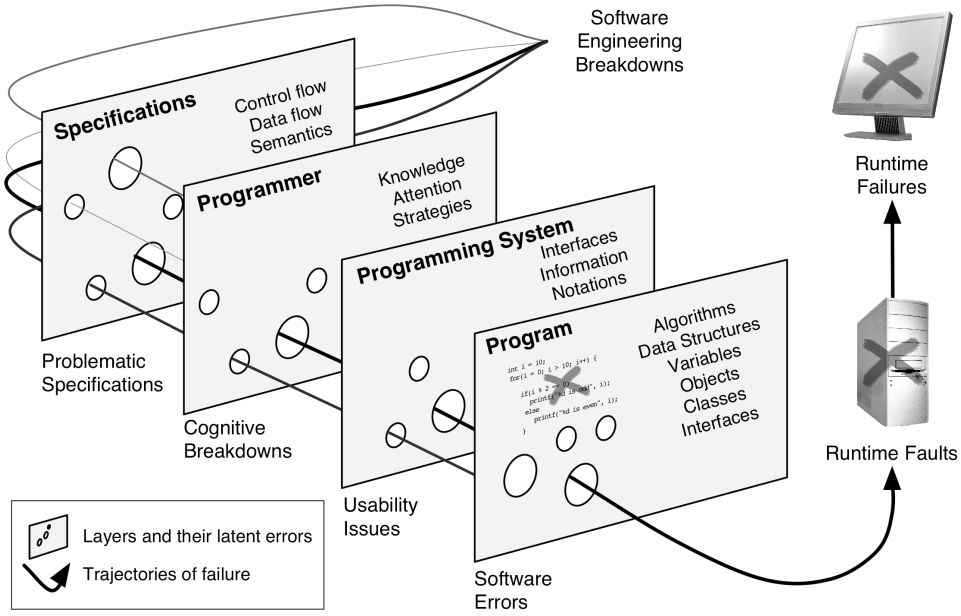

Figure 3. Dynamics of software error production, based on Reason's "Swiss-cheese" model of failure. Each layer has latent errors (the holes), predisposing certain types of failures. Layers also have defenses against failures (where there are no holes). Many layers of failure must occur for software errors to be introduced into code.

We apply these ideas to software engineering in Figure 3. In the figure, we can see there are many layers, and each layer has its own type of latent errors and defenses. For example, specifications, which have information such as control and data flow semantics, are meant as high-level defenses against software errors. However, specifications are often ambiguous, incomplete, or even incorrect, predisposing programmers to

misunderstandings about the true requirements. Therefore, 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, with knowledge and expertise as defenses, are still prone to a host of *cognitive breakdowns* that may lead to software errors (we will explain these in the next section). By educating programmers, we can reduce the number of software errors introduced into code. Components of a programming system, such as the compilers, libraries, programming languages, and environment, may have many built-in defenses against software errors, but may also have usability issues that predispose the programmer to cognitive breakdowns. For example, compilers defend against syntax errors, but in displaying confusing error messages, may misguide programmers in repairing the syntax errors.

It is important to note that the "holes" in each of these layers are not static; latent errors may only cause failure under particular circumstances, just like a program may only fail with particular input. Furthermore, because programmers adapt to changes in their environment, properties of the programming system can effectively open and close holes in programmers' cognition.

1.3.2 Skill, Rule, and Knowledge Breakdowns

Within this broad view of failure, we 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. Under most circumstances, these

"default" behaviors are sufficient; however, under exceptional or novel circumstances, they can lead to error. In programming, this means that when solving problems, programmers tend to prefer strategies that have been successful in the past. Most of the time, these default strategies are successful, but sometimes they break down—hence the term *cognitive breakdowns*.

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 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 in a particular programming environment, or compiling a program by pressing a button in an IDE. These are practiced, routine, automatic 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. These breakdowns can lead to errors.

Table 3 lists Reason's two categories of skill breakdowns. *Inattention* breakdowns are due to 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 a smoke detector goes off. When he returns to his task after the interruption, he fails to complete the increment statement, introducing an error. Inattention breakdowns are not due only to interruptions: they may also occur because of the intrusion of strong habits. For example, consider a programmer that tends to save modifications to a source file after every change, so that important modifications are not lost. At one point during modifications, 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 are the opposite of inattention breakdowns: they are due to attending to routine actions when it would have been best to "leave it in 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 looks two lines down from where he actually was. He falsely assumes that the statements above were already coerced, which causes him to neglect two variables, and thus introduce two software errors into his code.

Table 3. Types of skill breakdowns, adapted from Reason [20].

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

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, when expert C programmers need to print a list of values from an array, they might employ the rule, "If something needs to be done to a list of values, type *for(int i = some_initial_value; i < some_terminating_value; i++)* and then choose the initial and terminating values." These rules are much like the concept of *programming plans* [21], and are thought to underlie the development of programming expertise [22].

Table 4 lists Reason's two categories of rule breakdowns. The first category is *wrong rule*. Because rules are defined 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 incorrect in some circumstances, and thus a rule that is perfectly reasonable in one context may be selected in an inappropriate context. For example, one common error in Visual Basic.NET is that programmers will use the "+" operator to add numeric values represented in two strings, not realizing that the variables are strings. 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 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 I need to share data amongst all of the event-handlers, create a global variable on the form" is appropriate, but using this rule in object-oriented languages is inappropriate. Ko and Uttl demonstrated that in learning an unfamiliar statistical programming system, expert programmers' existing rules for other languages negatively influenced their acquisition of rules for the new system [23]. In their studies, experienced Java programmers expected loops to operate on a single data element at a time, when in fact they operated on a whole set of cases at once. In a study of Pascal, Shackelford studied novices use of three types of *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, but critically different problems [24].

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

The second type of rule breakdowns are *bad rules*, where a rule has problematic conditions or actions. These rules are come from learning difficulties, a lack of experience, or a lack of understanding about a program's semantics. For example, Perkins, Fay and Soloway 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]. The classifications listed in Table 1 illustrate many examples of bad rules. Not knowing the language syntax—in other words, not encoding or inaccurately encoding properties of the language—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 [25].

In *knowledge-based* activities, 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. Knowledge-based activities rely heavily on the interpretation and evaluation of models of the world (in programming, models of a program's semantics) and are therefore 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 people form. One important limitation on human cognition is *bounded rationality* [26]: 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: (1) evaluate information that is easily accessible in the world or in the head (the *availability heuristic*); (2) evaluate information that is easy to evaluate (*selectivity*); and (3) only evaluate as much as is necessary to form a plan of action (*biased reviewing*).

Because of the complexity of programming activity, bounded rationality shows up in many programming tasks. For example, Vessey argues that debugging is difficult because the range of possible errors causing a runtime failure is highly unconstrained and further complicated by that fact that multiple independent or interacting errors may be to blame

[27]. Gilmore points out that, because of their limited cognitive resources, programmers generally only consider a few hypotheses of what software error 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 [28]. For example, in response to a program displaying an unsorted list because the sort procedure was not called, a programmer might instead decide the error was an incorrect swap algorithm, and attempt to modify the already correct swap code.

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 heuristic	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

Another source of knowledge breakdowns is *faulty models of the problem space*. For example, human cognition tends to sees illusory correlations between events (*illusory correlation*), and even develops rationalizations to defend the belief, even in the face of more accurate observations (*confirmation bias*). These reasoning problems lead to oversimplified models of the problem space. Human cognition also evaluates knowledge in biased ways. For example, individuals display *overconfidence*, giving undue faith to the correctness and completeness of their knowledge. This results in strategies that are based on incomplete analyses. In programming activity, there are a number of sources of

overconfidence. For example, spreadsheet programmers exhibit overconfidence in the correctness of their spreadsheet's formulas [29]. Corritore and Wiedenbeck's studied corrective maintenance activity, finding that programmers' overconfidence in the correctness of their mental models of a program's semantics was often the cause of programmer's modification errors [30].

2 A Framework for Studying the Causes of Software Errors

One reasonable outcome of our review of past research could be a list of heuristics for preventing software errors in programming systems. If the causes of software errors were due entirely to *programmers'* inadequacies, this might not be so difficult to produce: as we saw in the previous section, there are a limited number of reasons that human cognition fails. Unfortunately, we have also seen that the causes of software errors are due to somewhat subtle issues: for example, hidden or ambiguous signs in programming environments, unfortunately-timed interruptions, or slight misunderstandings about a language construct's runtime behavior. Without knowledge of these details, it is difficult to suggest how programming systems might be redesigned to prevent breakdowns.

Therefore, rather than provide guidelines for preventing cognitive breakdowns, we propose a framework for describing cognitive breakdowns and the mechanisms by which they are related.

2.1 A Framework for Describing the Causes of Software Errors

Our framework integrates four aspects of previous research:

1. *Section 1.2*. Programmers perform three types of *programming activities*: specification activities (involving design and requirements specification),

implementation activities (involving the creation of code), and runtime activities (involving testing and debugging).

2. *Section 1.2*. Programmers perform six types of *actions* while interacting with a programming system's interfaces: design, creation, reuse, modification, understanding, and exploration.
3. *Section 1.3.1*. Chains of failure in specification, programmer's cognition, and programming systems can cause software errors.
4. *Section 1.3.2*. Programmers are prone to *skill*, *rule*, and *knowledge breakdowns* both because of inherent, internal cognitive limitations and external properties of the world.

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 introducing software errors into code.

These ideas map directly to elements in our framework, which is portrayed in Figure 4. The three grey regions, stacked vertically, denote specification, implementation, and runtime activities. The four columns contain the possible components of a breakdown within an activity. For example, in specification activities, a breakdown may involve of one of three types of breakdowns, one of three types of actions, one of three types of interfaces, and one of two types of information (therefore, the framework can describe 3 x 3 x 3 x 2 = 54 types of specification breakdowns). The available actions, interfaces, and

information 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 created by following the arrows in Figure 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 specifications using a diagram can cause a knowledge breakdown in implementing code." The framework allows all relationships *between* and *within* each activity. During specification, for example, problems in creating specifications can cause problems modifying them. Or, between specification and implementation activities, specification breakdowns can cause implementation breakdowns; but implementation breakdowns can also cause specification breakdowns, since in understanding code, programmers may change their mental models of specifications.

In addition to allowing "can cause" relationships within and between activities, the framework also allows relationships between software errors, runtime faults, runtime failures, and other breakdowns. For example, software errors can cause implementation breakdowns before causing a runtime faults when a programmer makes a variable of Boolean instead of integer type and tries to increment it. Also, a runtime fault or failure can cause debugging breakdowns once a programmer notices them.

While our framework suggests many links between breakdowns, it makes no assumptions about their ordering. High-level models of software development, such as the waterfall or extreme programming models, assume a particular sequence of specification, implementation, and debugging activities. Low-level models of

programming, program comprehension, testing, and debugging assume a particular sequence of programming actions. Our model hopes to describe software errors introduced in any of these processes.

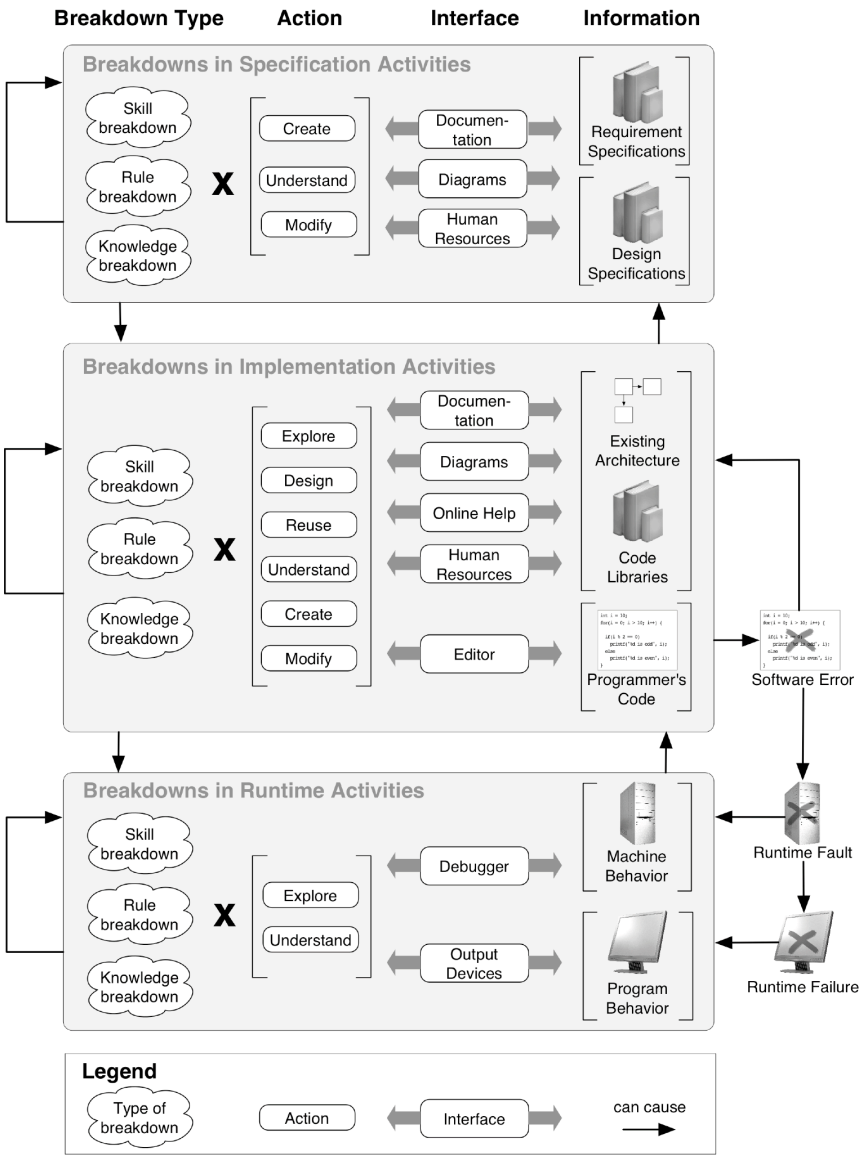

Figure 4. A framework for describing the causes of software errors based on chains of cognitive breakdowns. Breakdowns may occur in specification, implementation, and runtime activities. A single breakdown consists of one component from each column, within an activity. The cause of a single software 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.

2.2 An Example Chain of Cognitive Breakdowns

To illustrate how these chains of breakdowns occur, consider the true scenario illustrated in Figure 5. A programmer had little sleep the night before, which causes a skill breakdown in implementing the swap algorithm for a recursive sorting algorithm; this causes a repeated variable reference. At the same time, an inadequate knowledge breakdown in understanding the algorithm's specifications causes a knowledge breakdown in implementing a statement in the recursive call; this causes an erroneous variable reference. When he tests his algorithm, the sort fails. When observing the failure, the programmer has a rule 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 knowledge breakdown in understanding the runtime failure: he forms an incorrect hypothesis about the cause of the failure, and neglects to consider other hypotheses. This causes a knowledge breakdown in understanding the cause of the runtime fault, leading him to focus on the wrong code. This invalid hypothesis causes a knowledge breakdown in modifying the recursive call, and the programmer causes the algorithm to infinitely recurse.

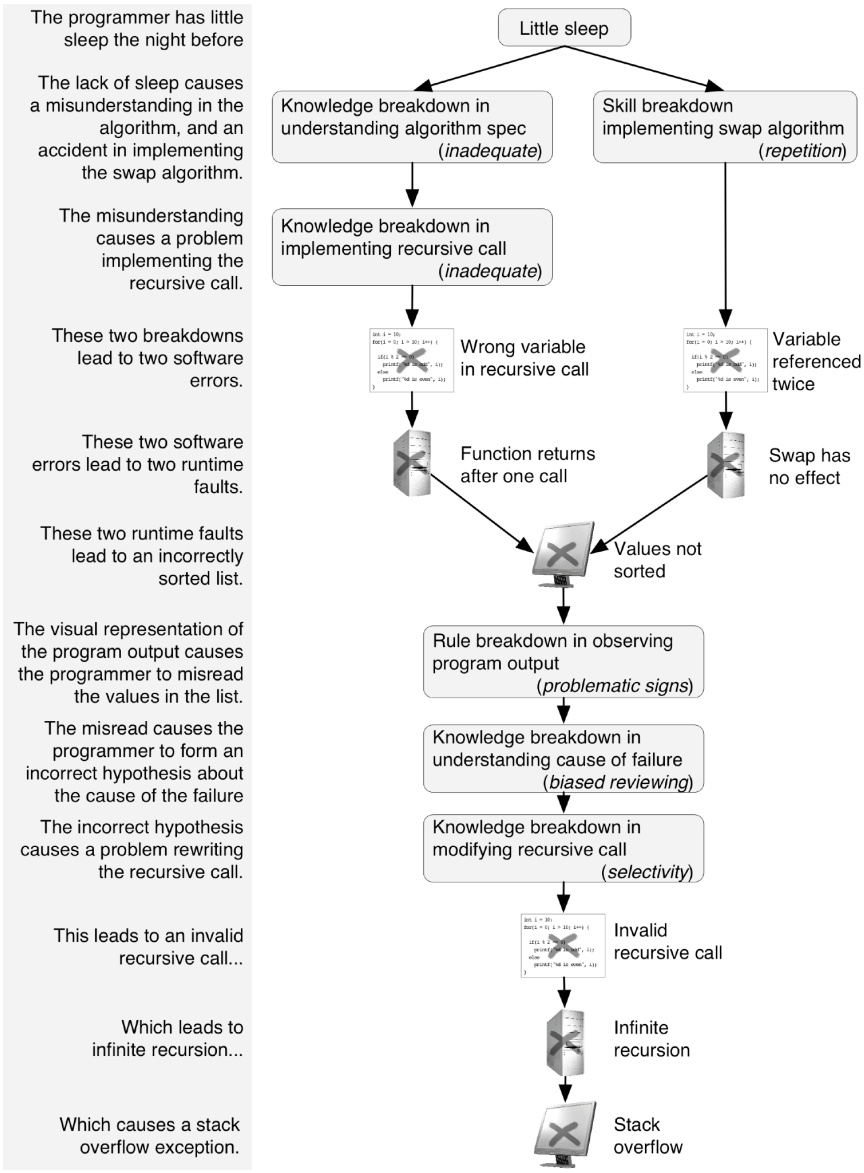

Figure 5. An example of a chain of cognitive breakdowns, where a programmer has difficulty implementing a recursive sorting algorithm.

3 An Empirical Methodology for Studying the Causes of Software Errors in Programming Systems

Our framework for describing the causes of software errors could certainly be used as a conversational tool, to support subjective discussion about the possible causes of software errors in a particular programming system. For example, "Interface A in the code editor might make programmers prone to skill-based breakdowns in cutting and pasting code, since it obscures part of the text that's being pasted."

In addition, the framework can also be used for more objective, empirical analyses of a programming system's error-proneness. The underlying assumption of the methodology is that *a programming system is prone to a subset of all possible chains of breakdowns described by the framework*. Therefore, the empirical goal in using our framework is to sample a large number of chains of cognitive breakdowns, and perform statistical analyses on the chains in order to find the most common causes and interfaces involved. By performing these analyses, researchers can get a more objective model of the error-prone aspects of the system, highlighting interfaces and actions that are involved in causing cognitive breakdowns. This allows designers of programming systems to set design priorities and provides detailed design knowledge for designing programming interfaces to prevent cognitive breakdowns.

In this section, we discuss a number of methodological issues in performing such empirical studies. To get a general sense for the methodology, here is the overall procedure:

1. Design a programming task that requires the use of the programming interfaces under study.
2. Observe and record programmers working on the task, using *think-aloud* methodology to capture their decisions and reasoning.
3. Using the recorded actions and verbal data, reconstruct chains of cognitive breakdowns starting from software errors or runtime failures.
4. Analyze the chains of cognitive breakdowns for patterns and relationships.

3.1 Sampling Chains of Cognitive Breakdowns

Four components of a breakdown must be sampled: the type of breakdown, the action performed, 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, etc.), the *actions* she performs using these interfaces (editing code, shell commands, etc.), and the *information* that she is acting upon (code, makefiles, executables, etc.). The most reliable ways to record observable actions are to videotape programmers working with a system and record the contents of the screen using video capture software. While it is also useful to instrument a programming system to record programmers' actions, this often constrains the level of abstraction at which actions are recorded. For example, an environment instrumented to record programmers' modifications to code would not capture the interface used or the interactive problems encountered along the way.

The only unobservable component of a breakdown is its *type*. Obviously, we cannot determine programmers' decisions and reasoning simply by watching them work. Instead, we use *think-aloud* methodology to elicit programmers' self-reports of their decision-making. Programmers' verbal utterances are then used to ask *deductive questions* about some event. 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 would deduce that he had a knowledge breakdown due to biased reviewing. This is because he was in knowledge-based cognitive activity, and only considered one course of action.

To help answer deductive questions about breakdowns, we summarize the types of skill, rule, and knowledge breakdowns in Table 6 and the cognitive activities in which they occur. This table can be used to find an appropriate answer for each deductive question. 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. Nevertheless, recording all of the *possible* causes can be useful and informative as well.

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*)

In addition to recording a breakdown's type, action, interface, and information, it is also helpful to record details about these four components. For example, if the programmer exhibited overconfidence, what were they overconfident about? Or, if the programmer is

inundated with signs and does not attend to the right one, what other signs were present in the programming environment and why was the important sign not attended to?

3.2 Reconstructing Chains of Cognitive Breakdowns

We also use a deductive approach to reconstructing *chains* of cognitive breakdowns. To reconstruct a chain, one asks deductive questions starting from a runtime failure, and continuing until no other cause can be found.

This process is portrayed in Figure 6, which reconstructs the chain presented in Figure 5 using hypothetical observations from videotape. From the failure, we ask the deductive question, "What caused the stack overflow?" and proceed to deduce to the cause of the software error. Deducing the chain of causality from the failure 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.

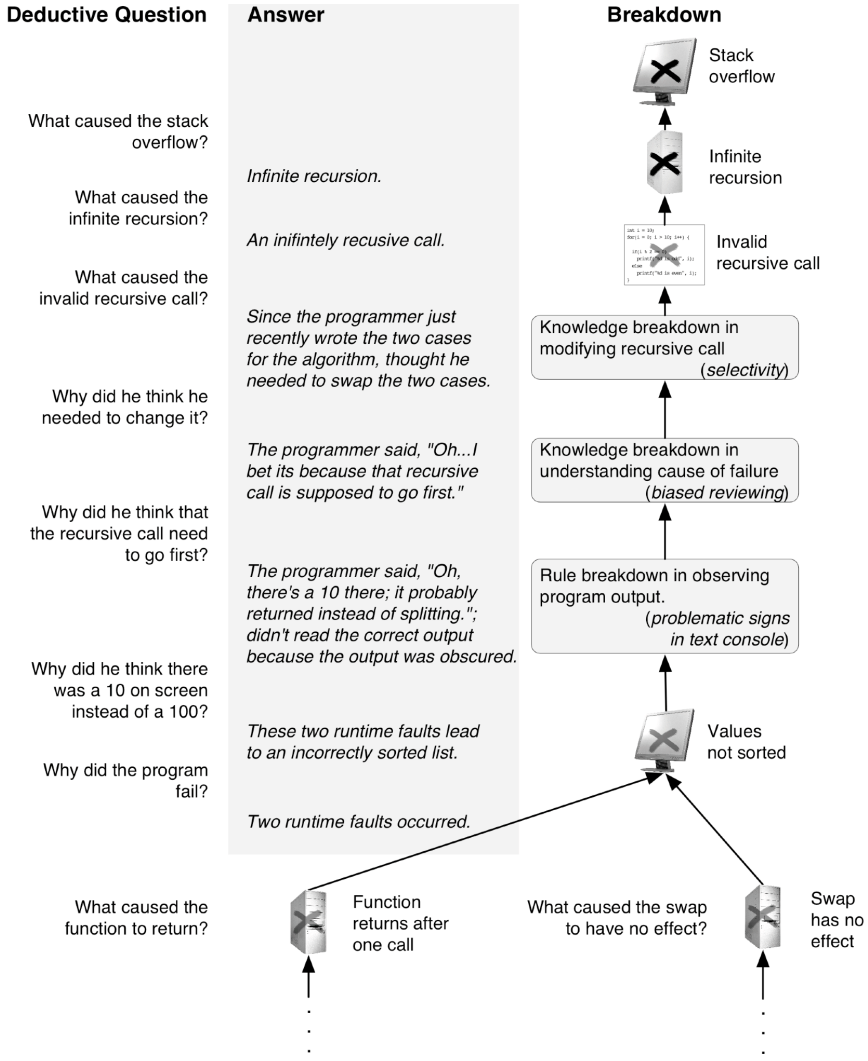

Figure 6. Deductively reconstructing the causal chain of breakdowns represented in Figure 5, using observations of the programmer's actions from videotape.

Once the software errors leading to failure have been deduced, once must deduce the cognitive breakdowns causing the error using the techniques described in the previous section. For example, in Figure 6 we ask, "What caused the invalid recursive call?" We rely on the fact that the programmer said "Oh, I bet its because that recursive call was

supposed to go first" and then proceeded to move the recursive call in his code (had the programmer said nothing about his actions, there would have been many possible cognitive breakdowns to blame, but none with supporting evidence). We then proceed to ask deductive questions about each successive breakdown, until no cause 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 Figure 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 failures (due to multiple runtime faults), at errors (due to multiple cognitive breakdowns), and at breakdowns (due to multiple external events, such as interface problems or interruptions).

Because software errors are defined relative to design specifications, it is only possible to start reconstruction from software errors when the experimenter knows the program's design specifications. Without this knowledge, it is difficult to tell what constitutes an error. In end-user programming contexts, errors are particularly difficult to detect: because the design specifications are exclusively in the programmer's head, the roles and relationships between code may change at any time. The only reliable way to identify an error in such cases is when the programmers explicitly identify their approaches to designing a program (thereby defining their design specifications).

3.3 Task Selection and Participants

A critical choice in designing an empirical study using our framework is the programming task. Clearly, some programming tasks involve more code creation than debugging, and more uses of some interfaces in a programming environment than others. Therefore, the decision must be based on the intents of the study. In addition to considering the purpose of the study, the complexity of the task should be considered as well. If the program used in the task is very complex, it may be difficult to determine what code is erroneous, as defined by the specifications.

Participants should be recruited based on the difficulty of the task. Furthermore, participants should have similar programming experience in order to ensure they have similar breakdowns. Because our methodology analyzes the causes of errors, and not participant variables such as performance or learning, experiments need only recruit as many programmers as is necessary for a sufficient number of errors to analyze.

3.4 Think-Aloud Guidelines

There are a number of caveats in using *think-aloud* methodology to study the causes of software errors. The original rationale for think-aloud methodology from Ericsson and Simon's "Protocol Analysis: Verbal Reports as Data" [31] was that the only verbal data that can be collected reliably, without interfering with task performance, is that which does not require additional attentional resources to verbalize. Ericsson and Simon, and others, have demonstrated that problem-solving tasks that are describable in terms of verbalizable rules and generally without external, time-critical factors, fall in to this

category. 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 is an "interested learner." These roles should be defined explicitly and maintained throughout the experiment.
- 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, because in asking, they may bias participants' explanations, or elicit fabricated explanations.

By following these guidelines, participants' verbal utterances should be a valuable and reliable indicator of the type of breakdowns programmers have—but only as long as verbal data is analyzed in a reliable way. Because verbal utterances must be interpreted, errors or disagreements in these interpretations are inevitable. We recommend testing the

reliability of interpretations by having multiple individuals reconstruct chains independently, and then checking for agreement in the types and sequence of cognitive breakdowns.

3.5 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 about cognitive breakdowns.

- 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 instigate chains of further breakdowns?

With knowledge about the cognitive breakdowns that occur in a system, it is also important to consider how these chains of breakdowns are related to the software errors they cause:

- What types of cognitive breakdowns are most prone to software errors?
- What properties of interfaces are responsible for software errors?
- How long are the chains of breakdowns that cause software errors?
- Which software errors tend to cause further errors?
- What are the root causes of breakdowns, for which no other causes can be found?

4 Causes of Software Errors in the Alice Programming System

In this section, we present a case study of the causes of software errors in the Alice programming system. We describe our experiences in the hopes that other researchers can apply our methodology to new programming systems and interfaces.

4.1 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 programming environment, seen in Figure 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.

Figure 7. Alice v2.0, 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 code for the method being edited.

4.2 The Experiments

We had two goals in performing our studies of Alice:

1. Identify common cognitive breakdowns that were due mostly to the programmer's limitations, and what types of interfaces might help prevent these breakdowns.
2. Identify common cognitive breakdowns that were due mostly to the programming environment's interfaces, and how they might be redesigned to help prevent these breakdowns.

To answer these questions, we performed two studies of Alice in widely different contexts. First, we studied 3 Alice programmers who had been using Alice for six weeks in the "Building Virtual Worlds" course offered at Carnegie Mellon. These programmers were using Alice to prototype complex interactive worlds designed to be entertaining, and thus their requirements were constantly changing.

The second study was of 4 novice Alice programmers, and was performed in a lab setting. Programmers were asked to create a simple Pac-Man game with one ghost, four small dots, and one big dot (as seen in Figure 7). After a 15-minute tutorial on how to create code, methods, and events, programmers were given the requirements in Table 7.

Table 7 also lists the language expertise of the seven participants from both studies. Both observational studies used the methodology described in Section 3: programmers were asked to think aloud about their programming decisions and were videotaped while they worked. The experimenter used the phrases "And now?" and "Please continue" thirty seconds after silence, to remind the programmers to think aloud.

Table 7. For both experiments, programmers' self-rated language expertise and the tasks that they performed during our observations.

Experiment	ID	Language Expertise	Programming Tasks
"Building Virtual Worlds" study	B1	Average C++, Visual Basic, Java	• 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	• Randomly resize and move 20 handlebars in a subway train
	B3	Above average C, Java	• Import, arrange, and programmatically animate objects involved in camera animation.
Pac-Man study	P1	Above average Java, C	• Pac must always move. His direction should change in response to arrow keys. • Ghost must move in randomly half of the time and towards Pac the other half. • If Ghost is chasing and touches Pac, Pac must flatten and stop moving forever. • 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.
	P2	Below average C++, Java	
	P3	Above average Java, C++	
	P4	Above average Visual Basic	

4.3 Analyses and Results

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 study errors that did *not* cause runtime failures, because we could not identify them: as discussed in the previous section, 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.

Each of the videotapes was analyzed for runtime failures and all of the programmers' verbal utterances were transcribed, along with timestamps for each. For each failure identified, we reconstructed the chain of cognitive breakdowns leading to it using the techniques in Section 3. One of the chains is depicted in Figure 8. In the figure, the

instigating breakdown in creating the specifications for the Boolean logic led to a knowledge breakdown implementing the logic, which led to two errors. These errors led to a fault and failure, and further breakdowns.

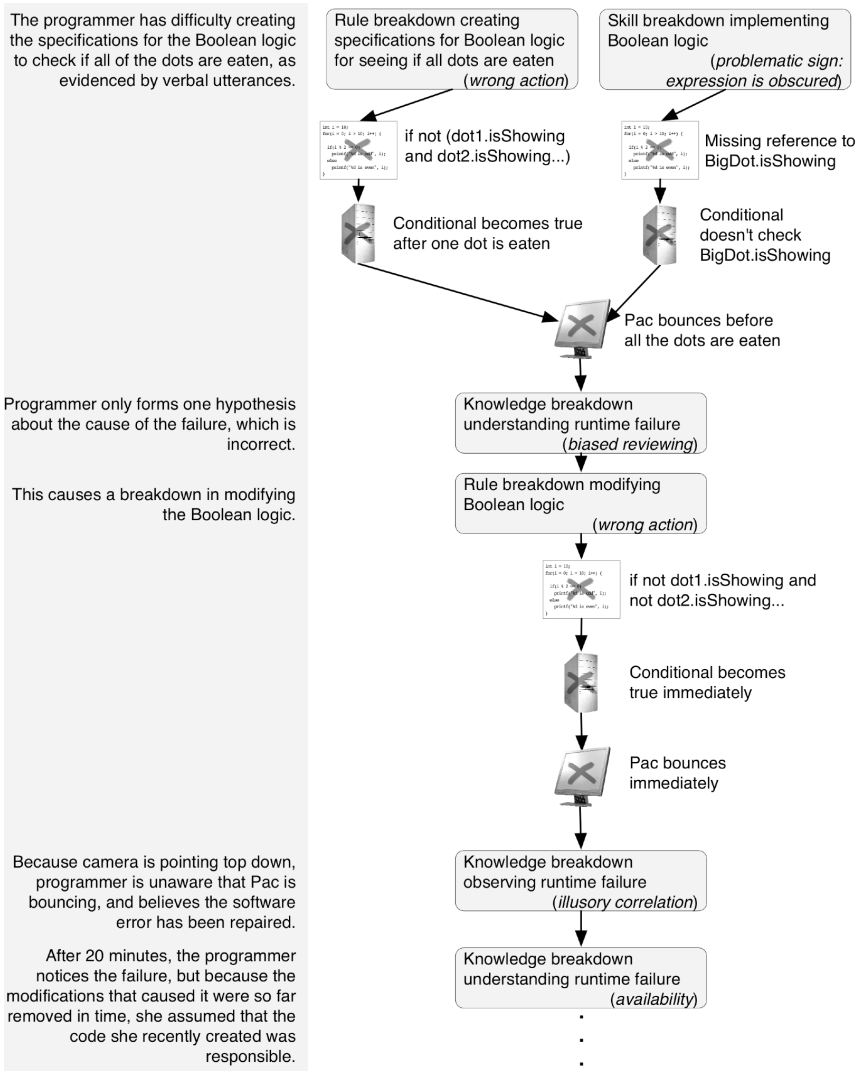

Figure 8. A segment of one of P2's cognitive breakdown chains. The last breakdown shown here did not cause further breakdowns until 20 minutes later, after the camera position made it apparent that Pac was jumping.

4.3.1 Overall Statistics

Over 895 minutes of observations, there were 69 root breakdowns (breakdowns with no identifiable cause) and 159 total breakdowns. These caused 102 errors, 33 of which led to one or more new errors. The average chain had 2.3 breakdowns (standard deviation 2.3) and caused 1.5 errors (standard deviation 1.1). Table 8 shows the proportions of time programmers spent programming and debugging. On average, 46% of programmers' time was spent debugging (and thus a little more than half was spent implementing code). Looking at the students in the "Building Virtual Worlds" class, whose code was more complex, we see that the length of their chains of cognitive breakdowns were longer than those programmers implementing the Pac-Man game, suggesting that the causes of their errors were more complex.

Table 8. Programming and debugging time, and errors, breakdowns, chains, and chain length by programmer.

ID	Programming Time	Debugging Time		Number of Errors	Number of Breakdowns	Number of Chains	Average Chain Length
	minutes	*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)

The total proportions of knowledge, rule, and skill breakdowns were similar, but proportions of activities were not: 77% of breakdowns were in implementation activity, and tended to be skill and rule breakdowns in implementing and modifying artifacts and knowledge breakdowns in understanding and implementing artifacts; 18% of breakdowns were in runtime activity, and tended to be knowledge or skill problems in understanding failures and faults. The root breakdowns of most chains were knowledge breakdowns understanding runtime failures and runtime faults and skill and rule breakdowns

implementing code. Table 9 shows which aspects of the programming system 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 the observations were of programmers completely new to the Alice programming system.

Table 9. Frequency and percent of breakdowns and errors by type of information and the average debugging time for errors in each type of information.

Type of Information	Breakdowns		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%	-	-	-
Style-specific	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%	-	-	-

Table 10 shows the number of software errors and time spent debugging by problem and action. Most 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 strategic problems reusing and knowledge problems understanding artifacts.

Table 10. Software errors and debugging time by cognitive breakdown type and action. Only actions causing errors are shown.

Breakdown	Action	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)

4.3.2 *Significant Causes of Software Errors in Alice*

There were four major causes of software errors in the studies. In each case, the Alice programming environment 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). In particular, there were two types of breakdowns. Most cases were rule breakdowns implementing complex Boolean expressions, because of bad rules. 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...)*" This confirms earlier studies by Pane showing that creating Boolean expressions is of considerable difficulty [33]. In other cases, whether or not they had created a correct expression, programmers suffered from rule breakdowns in implementing the expressions because of problematic signs in the Alice environment. When dropping *and* and *or* operators onto code, they were not sure which part of the expression they were affecting, often because it was off-screen or ambiguous.

With so many software errors introduced because of the implementation breakdowns, the 18% of debugging breakdowns 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 very limited 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 code unintentionally, and because the structure of the code was not clear, did not realize they had introduced software errors during modification.

A final source of software errors, largely independent of the cycles of breakdowns described above, were the 7% of reuse breakdowns. These were rule breakdowns in reusing code via copy and paste, because of 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, the property to coerce was off-screen, causing the programmer to not change the reference, and thus

introducing a software error. These errors were very difficult to debug, because of knowledge breakdowns in understanding their code due to overconfidence in the copied code's correctness. Thus, when programmers attempted to determine the cause of their program's failure, their hypotheses were instead centered on recent changes (knowledge breakdowns due to the availability heuristic). Furthermore, these errors caused complex, unpredictable runtime interactions. Because these reuse errors were so difficult to debug, few programmers ever found their error.

We summarize these trends in the causes of software errors in the model shown in Figure 9. By counting the frequency of specific segments in chains of breakdowns, we highlight the common links between particular types of breakdowns, and how they are related to the introduction of software errors.

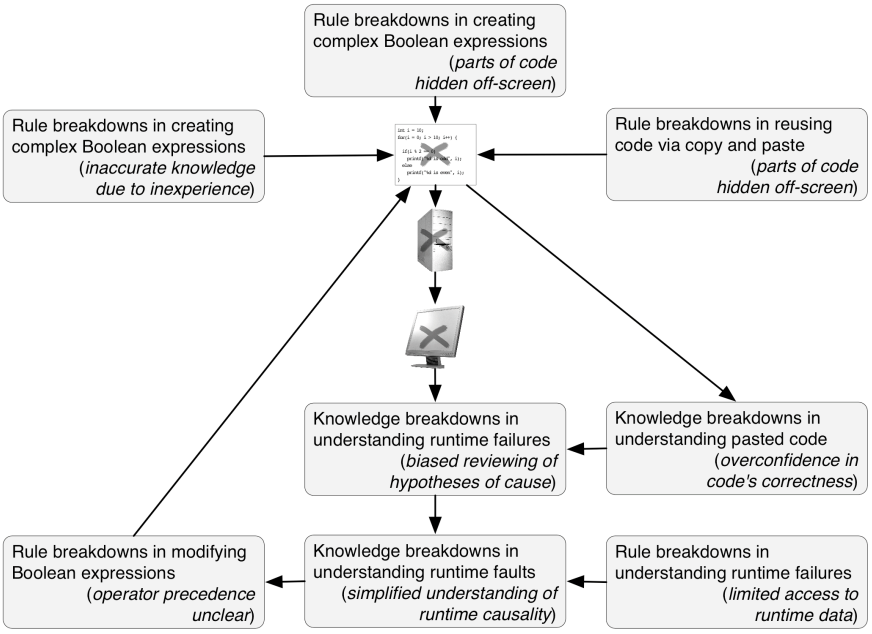

Figure 9. A model of most of the causes of software errors in Alice during programmers' tasks, based on analyses of the chains of breakdowns from the two observational studies.

4.4 Experiment Discussion

Our model of the software errors in Alice is of significant value in understanding the 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. However, the impact of these early errors was compounded by difficulties with debugging and modifying 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 because of modification breakdowns. Therefore, even in small tasks and simple environments,

there are complex relationships between software errors, the programming environment's interfaces, and the programmer's cognition and experience.

With a better understanding of the causes of software errors in Alice, there is significant opportunity to prevent these common breakdowns. We have since focused on preventing the knowledge breakdowns in understanding runtime faults and failures, with a debugging tool called the *Whyline* [34]. The argument behind the Whyline is that debugging tools should directly support programmer's formation of a hypothesis about the causes of a runtime failure. If they do not, programmers will have a weak hypothesis about the cause of a failure due to biased reviewing, and any implicit assumptions about what did or did not happen at runtime will go unchecked, as they did in these studies.

Therefore, the Whyline allows programmers to ask questions explicitly about their program's failure, preventing programmers from hypotheses about their program's runtime behavior altogether. By analyzing the programmers' explicit questions in these two studies, we found that programmers' questions at the time of failure were one of two types: *why did* questions, which assume the occurrence of an *unexpected* runtime action, and *why didn't* questions, which assume the absence of an *expected* runtime action. There were three possible answers:

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.

Based on these observations, we designed the Whyline to allow *why did* and *why didn't* questions about program's behavior. For example, had the Whyline been available when the programmer in Figure 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 of 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 user studies of the Whyline's usability [34]. By comparing six identical debugging scenarios from user tests with and without the Whyline, we 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 improvements are the direct result of preventing the biased reviewing breakdowns identified in our studies.

5 Discussion

We believe our framework and methodology can support reasoning about software errors, the study of software errors, and the design of error-robust programming systems.

5.1 Supporting Reasoning 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 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 Framework [6]. Cognitive Dimensions have been used to analyze the usability of many visual and professional programming languages [7, 8], but none have addressed the causes of errors. Future studies could identify relationships between dimensions of notations and the causes of programming errors. For example, consider *viscosity*, or, 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.

Another way in which our framework supports reasoning about software errors is in clearly identifying 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.

5.2 Supporting the Study of Software Errors

In addition to helping reason about software errors, our framework has a number of implications for the study of software errors and programming activity in general. For example, while von Mayrhauser and Vans' Integrated Comprehension Model [25] 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 [28]. 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.

As demonstrated in our studies of Alice, our methodology for studying software errors has the potential to uncover patterns of software error production in programming systems. Because the framework is descriptive, it supports the objective comparison of software errors within and between programming environments, programs, languages, tasks, expertise, and other important variables. 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 and incomparable results.

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 to methods that have such already proven useful in less-controlled settings, such as Contextual Inquiry [35]. In fact, our study of students in "Building Virtual Worlds" was very similar to industry settings: programmers were constantly interrupted, specifications were constantly changing, and programmer's attention was continuously divided among programming and numerous non-programming tasks. Despite these circumstances, there was little difficulty in collecting data. This is in stark contrast to empirical studies of programming that have been performed in the past, which require significantly more intervention to obtain reliable results.

5.3 Supporting the Design of Programming Systems

Our framework and methodology also has considerable potential to support the *formative* design of programming systems. By focusing on the limited number of failures in human cognition discussed in Section 1.3 and summarized in Table 6, a simple set of heuristics for designing error-preventing programming systems can be generated. We list ten such heuristics in Table 11.

Table 11. 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

These heuristics can provide simple guidance for designers of programming languages and systems. For example, consider applying heuristic 2 to a number of design situations. 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 consider heuristic 2 when considering operator overloading, since programmers unfamiliar with the particular semantics of an overloaded operator may make erroneous assumptions about the program's runtime semantics. 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 and bringing them to programmer's attention.

The framework and methodology can also be used directly as a formative design tool on paper and Wizard of Oz prototypes. There is no requirement that chains of breakdowns are reconstructed from interactions with a functional prototype. In fact, we performed such analyses on paper and Wizard of Oz prototypes of the Whyline, in order

to determine if the Whyline would help keep programmer's assumptions in check. The only requirement for using our methodology for non-functional prototypes such as these 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 actually implementing a system.

6 Conclusions

This paper presents a framework and methodology for studying the causes of software errors in programming systems. The framework is derived from past classifications of errors, prior studies of programming, and general research on the mechanisms of human error. Our framework is based on the idea of three types of cognitive breakdowns, and the formation of chains of cognitive breakdowns during programming activity. Our methodology is focused on sampling chains of cognitive breakdowns by observing programmers work and then reconstructing chains of breakdowns from programmers' actions for later analysis.

Our experiences with studying the causes of software errors in the Alice programming system suggest that our framework and methodology are straightforward and helpful in revealing the sources of errors and in helping design new tools. Because our framework provides a common vocabulary for reasoning about software errors, while supporting their description, prediction, and explanation, we encourage other researchers to apply our framework and methodology to their research.

7 Acknowledgements

We would like to thank the programmers in our user studies for their participation and the faculty and students of the Human-Computer Interaction Institute for their constructive comments and criticisms of this work in the past year. 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.

8 References

[1] J. Reason, *Managing the Risks of Organizational Accidents*. Aldershot: Ashgate, 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 and 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, and P. Polson, The Cognitive Walkthrough: A Practitioner's Guide, in *Usability Inspection Methods*, J. Nielsen and R. L. Mack, Eds.: John Wiley and Sons, Inc., 1994.

[5] J. Nielsen, *Usability Engineering*. Boston: Academic Press, 1993.

[6] T. R. G. Green, Cognitive Dimensions of Notations, in *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds. Cambridge, UK: Cambridge University Press, 1989,443-460.

[7] T. R. G. Green and 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, Workshop on the Psychology of Programming Interest Group, Bournemouth UK, 2001, 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, 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, and R. Pausch, Alice: Lessons Learned from Building a 3d System for Novices, Proceedings of CHI 2000, The Hague, The Netherlands, 2000, 486-493.

[11] J. R. Anderson and 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, and 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 and H. A. Peelle, Apl Learning Bugs, APL Conference, Washington, D. C., 1983, 11-16.

[16] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper, Bug Catalogue: I, Yale University, Boston, MA, Technical Report 286, 1983.

[17] D. N. Perkins and F. Martin, Fragile Knowledge and Neglected Strategies in Novice Programmers, Empirical Studies of Programmers, 1st Workshop, Washington, DC, 1986, 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, 86-112.

[20] J. Reason, *Human Error*. Cambridge, England: Cambridge University Press, 1990.

[21] J. G. Spohrer and E. Soloway, Analyzing the High Frequency Bugs in Novice Programs, Empirical Studies of Programmers, 1st Workshop, Washington, DC, 1986, 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] A. J. Ko and B. Uttl, Individual Differences in Program Comprehension Strategies in an Unfamiliar Programming System, International Workshop on Program Comprehension, Portland, OR, 2003, 175-184.

[24] R. L. Shackelford and A. N. Badre, Why Can't Smart Students Solve Simple Programming Problems?, *International Journal of Man-Machine Studies* (38), 1993, 985-997.

[25] A. v. Mayrhauser and A. M. Vans, Program Understanding Behavior During Debugging of Large Scale Software, Empirical Studies of Programmers, 7th Workshop, Alexandria, VA, 1997, 157-179.

[26] H. Simon, Rational Choice and the Structure of the Environment, *Psychological Review*, 63, 1956, 129-138.

[27] I. Vessey, Toward a Theory of Computer Program Bugs: An Empirical Test, *International Journal of Man-Machine Studies*, 30 (1), 1989, 23-46.

[28] D. J. Gilmore, Models of Debugging, *Acta Psychologica* (78), 1992, 151-173.

[29] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook, Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?, Conference on Human Factors in Computing, 1997, 22-27.

[30] C. L. Corritore and 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.

[31] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data.:*. Cambridge, MA: MIT Press, 1984.

[32] M. T. Boren and J. Ramey, Thinking Aloud: Reconciling Theory and Practice, *IEEEE Transactions on Professional Communication*, 43 (3), 2000, 261-278.

[33] J. F. Pane and 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, 157-164.

[34] A. J. Ko and B. A. Myers, Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, Conference on Human Factors in Computing, Vienna, Austria, 2004, (to appear).

[35] K. Holtzblatt and H. Beyer, *Contextual Design: Defining Customer-Centered Systems*. San Francisco, CA: Morgan Kaufmann, 1998.