Development and Evaluation of a Model of Programming Errors

Andrew J. Ko and Brad A. Myers
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
ajko@cmu.edu, bam+@cs.cmu.edu

Abstract

Models of programming and debugging suggest many causes of errors, and many classifications of error types exist. Yet, there has been no attempt to link causes of errors to these classifications, nor is there a common vocabulary for reasoning about such causal links. This makes it difficult to compare the abilities of programming styles, languages, and environments to prevent errors. To address this issue, this paper presents a model of programming errors based on past studies of errors. The model was evaluated with two observational of Alice, an event-based programming system, revealing that most errors were due to attentional and strategic problems in implementing algorithms, language constructs, and uses of libraries. In general, the model can support theoretical, design, and educational programming research.

1. Introduction

In the past three decades, there has been little work in classifying and describing programmers' errors. Yet, the work that has been done was largely successful in motivating many novel and effective tools to help programmers identify and fix errors. For example, in the early '80's, the Lisp Tutor drew heavily from analyses of novices' errors [1], and nearly approached the effectiveness of a human tutor. The testing and debugging features of the Forms/3 visual spreadsheet language [3] were largely motivated by the type and prevalence of spreadsheet errors [18].

Recently however, the event-based style has become widely used, and no comparable description and classification of errors common in event-based systems exists. Not only do expert programmers widely use Java and C# to create interactive and enterprise applications, but many end users use Macromedia's Director, Microsoft's Visual Basic, Carnegie Mellon's Alice [4], and other event-based languages to create domain-specific interactive applications. To complicate matters, there is no common vocabulary for discussing the distribution of errors within the event-based style, or for describing and comparing errors in other styles, languages, tasks, domains, and environments. This makes it difficult to analyze what aspects of event-based programming are difficult, and to design programming environments that help prevent errors.

To address this issue, we have developed a general model of programming errors that ties the cognitive causes of programming errors to specific errors, integrating prior research on causes and classifications of errors. In this paper, we describe the model, and evaluate it using two observational studies of Alice [4], an event-based programming system. The model was helpful in describing and explaining errors, as well as in eliciting design guidelines for new programming tools.

2. What is a Programming Error?

We must first decide on an appropriate definition of a programming error. In this paper, we use definitions of *error*, *fault* and *failure* from IEEE standard 610.12-1990. A *failure* occurs when a program's output does not match documented output requirements, or the programmer's mental model of output requirements. Failures are ultimately the result of a *fault*, which is a runtime state of a program that either is or *appears to be* incorrect (as in assuming a lack of output from a debugging print statement to mean the code was not reached). Faults occur as a result of *errors*, which are program fragments that do not comply with documented program specifications or the programmer's mental model of specifications (such as a missing increment or misspelled variable name). Failures are usually the first indication to the programmer that one or more errors exist in a program, although some errors are found before they cause failures, since they may be similar to errors already found or may be apparent when inspecting code. While a failure guarantees one or more faults exist and a fault guarantees one or more errors exist, errors do not always cause faults, and faults do not always cause failures.

Proceedings of the IEEE 2003 Symposia on Human Centric Computing Languages and Environments (HCC'03), Auckland, New Zealand, October 28-31, pp. 1-8.

3. Classifications and Causes of Errors

Past classifications of errors identify a variety of types and causes of errors in many languages, environments, and levels of expertise. Table 1 summarizes classifications often cited in the literature chronologically. The summary is meant to be a representative sample of past classifications, rather than an exhaustive list.

There are many interesting patterns in the studies: failures due to errors can occur at compile-time and run-time; a given error has many possible causes, including lack of knowledge and forgetting; errors are made during specification, implementation, and debugging activities; there are a variety of artifacts which are error prone.

Many of these patterns were found in empirical studies of programming activity, and generalized into models that are better able to suggest the causes of errors. For example, recent models of programming activity suggest that programmers form a mental model of a program's documented specifications [17]. This mental model may be insufficient because of a lack of domain or task knowledge [9], an inadequate comprehension of the specifications [5], or a poor description of a programs requirements. This lack of knowledge can cause programmers to make a variety of errors. For example, a programmer may intentionally sort a list in increasing order, forgetting that the specifications called for decreasing order. Of course, creating and modifying documented specifications may cause unforeseen problems, leading to errors as well.

Study Details	Bug / Error / Cause	Description	Authors' comments
Gould [10], novice, Fortran *1975*	Assignment bug	Errors assigning variables values	Requires understanding of language & behavior
	Iteration bug	Errors iterating	Requires only an understanding of the language
	Array bug	Errors accessing data in arrays	
Eisenberg [7], novice, APL 1983	Visual bug	Clustering semantically related parts of expression	
	Naive bug	Using branching & iteration instead of parallel processing	"…because of need to think step-by-step"
	Logical bug	Omitting or misusing logical connectives or relationals	
	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	Not foreseeing side effects of commands	"…failure to see the whole picture"
Johnson et al. [14], novice, Pascal *1983*	Missing	Omitting required program element	Errors have contexts: input/output, declaration, initialization and update of variables, conditionals, scope delimiters, or combinations of these contexts.
	Spurious	Including unnecessary program element	
	Misplaced	Putting necessary program element in wrong place	
	Malformed	Putting incorrect program element in right place	
Sphorer and Soloway [19]; novice, Basic *1986*	Data-type inconsistency problem	Misunderstanding differences between 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 problem	Appling natural language semantics to commands	
	Human-interpreter problem	Assuming computer has similar interpretation of code	
	Negation & whole-part problem	Difficulties with constructing logical Boolean statements	
	Duplicate tail-digit problem	Incorrectly typing constant values	
	Knowledge interference problem	Domain knowledge interfering w/ entering constants	
	Coincidental ordering problem	Malformed statements produced correct output	
	Boundary problem	Not anticipating problems with extreme values	
	Plan dependency problem	Unforeseen dependencies in program statements	
	Expectation & interpretation problem	Misunderstandings of the problem specification	
Knuth [15], in writing TeX in SAIL & Pascal *1989*	Algorithm awry	Improperly implemented algorithms	"method proved to be incorrect or inadequate"
	Blunder or botch	Accidentally writing code not to specifications	"not… enough brainpower left to get the…details"
	Data structure debacle	Errors using and changing data structures	"I did not preserve the appropriate invariants"
	Forgotten function	Missing implementation	"I did not remember to do everything I had intended"
	Language liability	Misusing or misunderstanding language/environment	"I misused or misunderstood the…language"
	Mismatch between modules	Imperfectly knowing specs, interface; reversed arguments	"I forgot the conventions I had built"
	Reinforcement of robustness	Not handling erroneous input	"I tried to make the code bullet-proof"
	Surprise scenario	Unforeseen interactions in program elements	"forced me to change my original ideas"
	Trivial typos	Incorrect syntax, reference, etc.	"although my original pencil draft was correct"
Eisenstadt [8], industry experts, COBOL, Pascal, Fortran, C *1993*	Clobbered memory bugs	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 bugs	Unanticipated case, wrong algorithm	
	Initialization bugs	Erroneous type or initialization of variables	
	Variable bugs	Wrong variable or operator used	
	Lexical bugs	Lexical problem, bad parse, ambiguous syntax	
	Language	Misunderstandings of language semantics	
Panko [18], novice, Excel *1998*	Omission error	"Facts to be put into the model…but are omitted,"	Quantitative errors: "errors that lead to an incorrect, bottom line value"
	Logic error	Incorrect algorithm or incorrectly implemented algorithm	
	Mechanical error	"Typing the wrong number…or pointing to the wrong cell"	
	Overload error	Working memory unable to complete task 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	

Table 1. Studies classifying errors, bugs and causes in various languages, expertise, and contexts, in chronological order.

Even if a programmer's specification knowledge is sufficient, a programmer may create errors while implementing a program's specification because of a problem with implementation knowledge and strategies, as in von Mayrhauser and Van's model [17]. Factors causing these errors may include working memory overload, as in Green's parsing-gnisrap model of programming [12], unfamiliarity with a programming language and environment [16], inadequate knowledge of programming concepts, algorithms, and data structures, or an inability to apply the appropriate knowledge or strategies [6, 19]. Implementation errors include simple syntax errors, malformed Boolean logic, and scoping problems. The space of implementation errors overlaps greatly with that of specification errors. For example, a programmer may *unintentionally* sort a list in increasing order, not because she misunderstood specifications, but because of an inadequate understanding of the algorithm.

Vessey demonstrates that programmers can create errors even in the process of testing and debugging [21]. Her model of debugging suggests that only after programmers observe a failure do they realize that one or more errors exist, and the range of possible errors causing the failure is highly unconstrained. Further complicating the situation is that a failure may be caused by independent or interacting errors. Other models of debugging [9, 17] suggest that as programmers try to close the gap between failures and errors, they may falsely determine faults and errors, possibly leading to erroneous modifications. For example, in response to a program displaying an unsorted list because the sort procedure was not called, a programmer might instead determine the error was an incorrect swap algorithm, and erroneously modify the swap code.

These studies are very good at describing specific situations in which errors can be created. However, models of error in human factors research can significantly enhance our ability to reason about programming errors in general. Most notable is Reason's latent failure model of error [20]. In his model, he introduces the concept of *breakdowns*, which are problems using knowledge and strategies. Reason argues that strengthened knowledge and strategies make breakdowns less likely, but problems with knowledge, attention, and strategies can cause cascading breakdowns, each breakdown making error more likely. He discusses three cognitive problems that lead to breakdowns:

- **Knowledge problems**: inadequate, inert, heuristic, oversimplified, or interfering content or organization.
- **Attentional problems**: fixation, loss of situational awareness, or working memory strain.
- **Strategic problems**: unforeseen interactions from goal conflict resolution or bounded rationality.

4. A Model of Programming Errors

We use Reason's model as a basis for our model of programming errors. In our model, breakdowns occur in specification, implementation, and debugging *activities*, and consist of a *cognitive problem*, an *action*, and an *artifact*. Cognitive problems are Reason's knowledge, strategic, and attentional problems discussed earlier.

Available actions depend on the type of artifacts being acted upon. Documented and mental models of specifications can be *created*, *understood*, and *modified*. The meanings of these actions are different for each artifact. For example, understanding a documented specification is a software engineering skill, while understanding a mental model of a specification involves recall and reasoning. Implementation artifacts such as algorithms, data structures, and style-specific artifacts (such as events in Alice) can be *perceived*, *understood*, *implemented*, *modified*, and *reused*. The meanings of these actions depend on the environment and language.

Failures, faults, and errors can be *determined*. Determining failure involves perceiving and understanding output; determining a fault involves searching for what faults caused a failure; determining an error involves searching for what error caused a fault. Failures, faults, and errors have visual representations, so they can also be *perceived*. For example, determining if a program failed to sort a list may involve visually perceiving the list on a display—whether this is easy or not depends on the representation. These representations also have an underlying meaning, thus failures, faults and errors can also be *understood*. For example, understanding an error may involve understanding language semantics. Understanding a fault involves understanding a runtime state.

Our model hypothesizes many causal links between breakdowns. Breakdowns in an activity can cause more breakdowns within the activity, because actions within an activity often depend on each other. For example, problems in creating specifications can cause problems modifying them, and problems understanding algorithms can cause problems in implementing them.

Breakdowns in an activity can also cause breakdowns in another activity. Specification breakdowns can cause implementation breakdowns (this is what software engineers aim to prevent). Implementation breakdowns can cause specification breakdowns, since perceiving and understanding code can change mental models of specifications. Errors can cause implementation breakdowns before causing faults or failures (as in a programmer making a variable of Boolean instead of integer type and trying to increment it). Errors, faults, and failures can cause debugging breakdowns, and debugging breakdowns can cause implementation breakdowns (since programmers can create errors while debugging).

While our model suggests many links between actions in 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, and low-level models of programming, program comprehension, testing, and debugging assume a particular sequence of programming actions. Our model hopes to describe errors created in any of these programming processes.

A diagram of our model appears in Figure 1. The grey regions denote programming activities. The elements in these regions represent possible breakdowns comprised of cognitive problems (left column), actions (center), and artifacts (right column). In the figure, '**x**' means that any problem can apply to any action, which can apply to any artifact. The arrows denote a "can cause" relationship. Note that the figure does not portray every detail. There are many types of knowledge, attentional, and strategic problems, as described earlier, and there are many ways to perform actions depending on the environment and language. The figure only intends to provide a small, standard vocabulary for categories of cognitive problems, actions, and artifacts.

In general, the model assumes that as programmers work, cognitive problems cause them to traverse these causal links, generating a chain of breakdowns that may lead to errors. To illustrate these traversals, imagine this scenario. A programmer has inadequate knowledge for understanding documented specifications for a recursive sorting algorithm. This *instigating breakdown* causes a strategic problem in implementing the swap algorithm, which causes an erroneous variable reference, and the sort fails. When noticing the failure, the programmer has attentional strain in determining the fault, which leads to inadequate knowledge for modifying the swap algorithm. The programmer reads the documented specification again to mend this, but has attentional fixation in understanding it and mistakenly modifies his mental specification of the algorithm's recursion. This leads to unforeseen strategic problems when later modifying the recursive call, eventually leading to infinite recursion.

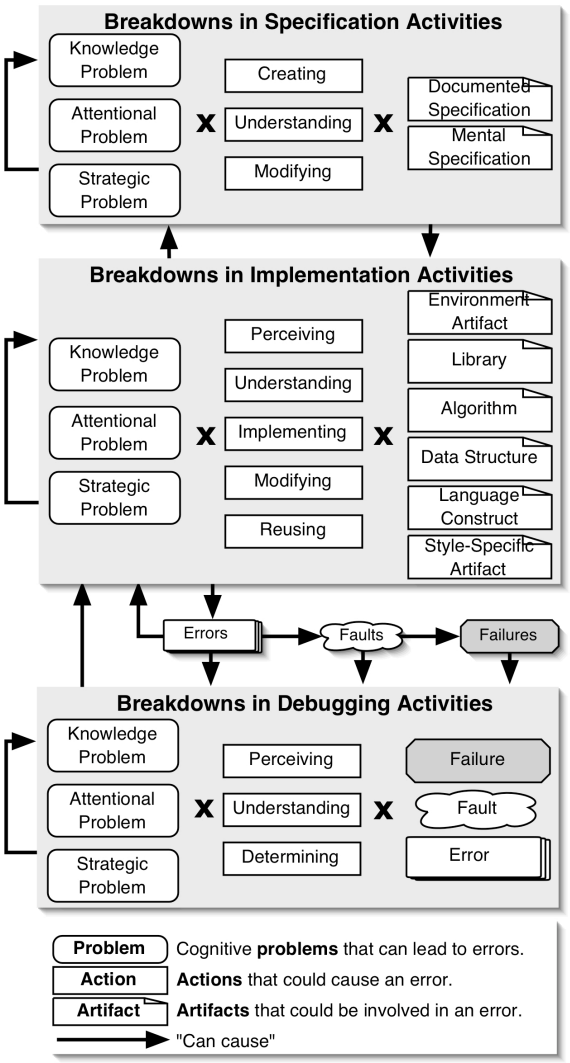

Figure 1. A model of programming errors, showing causal links between breakdowns in programming activities. Breakdowns are defined by combinations of cognitive problems, actions, and artifacts.

5. Evaluation

To evaluate the utility of the model, we performed two observational studies of programmers using the Alice 3D event-based programming system [4]. Alice provides objects (but does not support typical object-oriented features such as inheritance and polymorphism), provides explicit support for event-based constructs, and provides a drag-and-drop, structured editing environment that prevents syntax errors. A view of the Alice environment can be seen in Figure 2. See www.alice.org for details.

Figure 2. Alice v2.0: (1) objects in the world, (2) the 3D worldview, (3) events, (4) details of the selected object, and (5) the method being edited.

Experiment	ID	Hours Prog.	Language Expertise	Programming Tasks
Building Virtual Worlds students	B1	20	Ave. C++, Visual Basic, Java	Parameterize rabbit hop animation; make tractor beam catch rabbit; programmatically animate camera moving down stairs; prevent goat from penetrating ground after falling; play sound in parallel with character swinging bat.
	B2	10	High C++, Java, Perl	Randomly resize & move 20 handlebars using Jython, the Alice scripting language.
	B3	30	High C, Java	Import, arrange, & programmatically animate objects involved in camera animation.
Pac-Man participants	P1	5	High Java, C	(1) Make Pac-Man move perpetually & change direction when player presses arrow keys; (2) make ghost move in random direction half the time & towards Pac-Man otherwise; (3) if chasing ghost eats Pac-Man, make Pac-Man flatten & stop; (4) if Pac-Man eats big dot, make ghost run for 5 sec, then chase (5) if Pac-Man eats running ghost, make ghost stop for 5 sec & flatten (6) if Pac-Man eats all dots, make ghost stop & Pac-Man bounce.
	P2	< 1	Low C++, Java	
	P3	10	High Java, C++	
	P4	< 1	High Visual Basic	

Table 2. Total hours programmers spent programming the week of observation, self-rated language expertise, and tasks.

5.1 Experiment Descriptions

We were interested in describing programmers' breakdowns and errors, and the time spent debugging due to these errors. Though the studies involved a variety of tasks and expertise, they were not designed to control for these two factors. Both observational studies used the method of Contextual Inquiry [13]. As programmers worked, the experimenter tracked their goals, and asked programmers about their strategies and intents when not apparent. Programmers were also asked to think aloud about their programming decisions and were videotaped while they worked.

The first study involved 3 programmers enrolled in the "Building Virtual Worlds" course offered at Carnegie Mellon. In the course, students created complex Alice programs, working on a variety of programming tasks. Programmers had 6 weeks of prior experience with Alice.

The second study involved 4 programmers creating a simplified Pac-Man game with one ghost, four small dots, and one big dot, after a 15-minute tutorial on how to create code, methods and events. Programmers were given the objects and layout seen in Figure 2, and were instructed to follow the specifications listed in Table 2.

Table 2 shows the programmers' tasks and experience.

5.2 Experiment Results

Each of the videotapes was analyzed for programming tasks, task goals, goal start and stop times, strategies for achieving goals (as described by programmers), and results of using each strategy. From these transcripts, every breakdown involved in a failure was identified, along with the resulting errors, faults, failures, and times at which these occurred. From these breakdowns, the causal chain leading to each failure was constructed, like the one in Figure 3. In the figure, the instigating attentional breakdown in creating the specifications for the Boolean logic led to a strategic breakdown implementing the logic, which led to two errors. These errors led to a fault and failure, and further breakdowns. The analysis of the transcripts did not find chains that did not lead to failure, so we were unable to reason about breakdowns not involved in failures. Furthermore, due to a lack of data for comparing tasks and expertise, our analyses combined the datasets from the two studies.

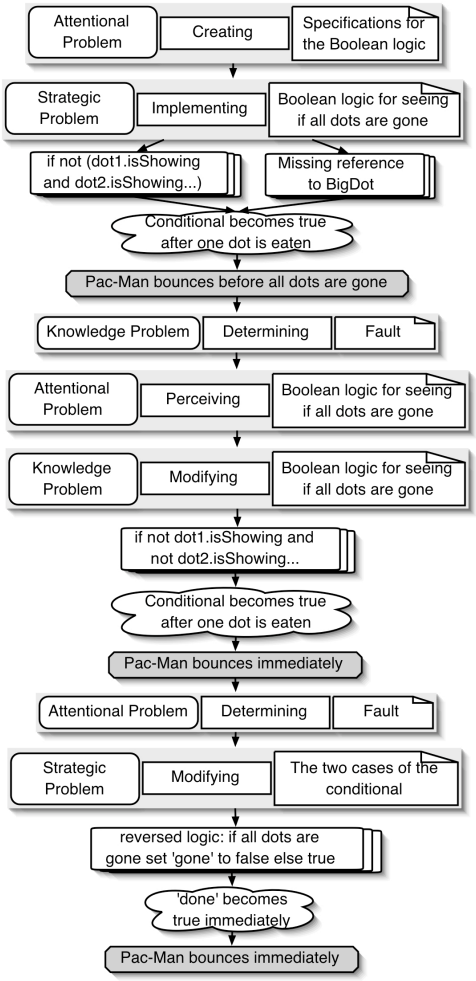

Figure 3. One of P2's longer breakdown chains, showing 6 breakdowns, 4 errors, 3 faults, and 3 failures.

There are many variables that could split such data, including activity, problem, action, artifact, task, environment, language, and expertise. There are also many useful measures: errors per minute, time spent debugging, percent of errors causing errors, number of instigating breakdowns, and chain length. For this paper, we were interested in a subset of these analyses.

Over 895 minutes of observations, there were 69 instigating 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).

Activity		Actions – *frequency (% of all 159 breakdowns)*							
	Problem	Create	Perceive	Understand	Implement	Modify	Reuse	Determine	**Total**
Specification	Knowledge	1 (.6) **1 (0.6)**		1 (.6) **1 (0.6)**					1 (.6) **1 (.6)**
	Attentional		n/a		n/a		n/a	n/a	
	Strategic	5 (3.1) **5 (3.1)**							5 (3.1) **5 (3.1)**
	Total	6 (3.8) **6 (3.8)**		1 (.6) **1 (0.6)**					7 (4.4) **7 (4.4)**
Implementation	Knowledge			16 (10.1) **17 (10.7)**	9 (5.7) **17 (10.7)**	0 (0) **6 (3.8)**	1 (.6) **3 (1.9)**		26 (16.3) **43 (27.0)**
	Attentional	n/a	1 (.6) **2 (1.3)**		10 (6.3) **15 (9.4)**	2 (1.3) **14 (8.8)**	3 (1.9) **4 (2.5)**	n/a	16 (10.1) **35 (22.0)**
	Strategic				14 (8.8) **23 (14.5)**	4 (2.5) **18 (11.3)**	1 (.6) **4 (2.5)**		19 (11.9) **45 (28.3)**
	Total		1 (.6) **2 (1.3)**	16 (10.1) **17 (10.7)**	33 (20.8) **55 (34.6)**	6 (3.8) **38 (23.9)**	5 (3.1) **11 (6.9)**		61 (38.4) **123 (77.4)**
Debugging	Knowledge							0 (0) **16 (10.1)**	0 (0) **16 (10.1)**
	Attentional	n/a	0 (0) **1 (0.6)**		n/a	n/a	n/a	0 (0) **12 (7.5)**	0 (0) **13 (8.2)**
	Strategic								
	Total		0 (0) **1 (0.6)**					0 (0) **28 (17.6)**	0 (0) **29 (18.2)**
Total	Knowledge	1(0.6) **1 (0.6)**		17 (10.7) **18 (11.3)**	9 (5.7) **17 (10.7)**	0 (0) **6 (3.8)**	1 (.6) **3 (1.9)**	0 (0) **16 (10.0)**	27 (17.0) **62 (38.4)**
	Attentional		2(1.2) **3 (1.9)**		10 (6.3) **15 (9.4)**	2 (1.3) **14 (8.8)**	3 (1.9) **4 (2.5)**	0 (0) **12 (7.5)**	16 (10.1) **47 (30.2)**
	Strategic	5 (3.1) **5 (3.1)**			14 (8.8) **23 (14.5)**	4 (2.5) **18 (11.3)**	1 (.6) **4 (2.5)**		24 (15.1) **50 (31.4)**
	Total	6 (3.8) **6 (3.8)**	2(1.2) **3 (1.9)**	17 (10.7) **18 (11.3)**	33 (20.8) **55 (34.6)**	6 (3.8) **38 (23.9)**	5 (3.1) **11 (6.9)**	0 (0) **28 (17.6)**	67 (42.1) **159 (100)**

Table 4. Frequency and percent of each combination of activity, problem, and action. Non-bold columns are instigating breakdowns and bold columns are all breakdowns, instigating or not. All percents are out of all 159 breakdowns.

Frequencies of breakdowns by activity, problems, and actions are shown in Table 4. Total proportions of knowledge, attentional, and strategic breakdowns were similar, but proportions of activities were not. Implementation breakdowns were 77% of breakdowns, and tended to be attentional and strategic breakdowns in implementing and modifying artifacts, and knowledge problems with understanding and implementing artifacts. Debugging breakdowns were 18% of breakdowns, and tended to be knowledge or attentional problems in determining errors and faults. Table 4 also shows frequencies of instigating breakdowns: most were knowledge problems understanding, and attentional and strategic problems implementing artifacts.

Table 5 shows that breakdown tended to be on algorithms, language constructs, uses of Alice libraries, and style-specific artifacts such as events. Note that about 19% of breakdowns were on faults and errors, and there were no breakdowns perceiving, understanding, or determining failures. Debugging times were highest for uses of libraries, algorithms, and language constructs.

Table 6 shows the errors and time spent debugging by problem and action. Most errors were caused by strategic problems implementing, modifying, and reusing artifacts (rather than understanding or perceiving artifacts). The variance in debugging times was high, and the longest debugging times were on strategic problems reusing and knowledge problems understanding artifacts.

Table 7 shows that 46 percent of programmers' time was spent debugging. Programming time was correlated with debugging time (r=.96, p<.001) and number of breakdowns (r=.88, p<.01). Errors were correlated with the number of breakdowns (r=.95, p<.001) and number of breakdown chains (r=.90, p<.01), and programming (r=.88, p<.01) and debugging time (r=.91, p<.01). Number of breakdowns was correlated with programming (r=.88, p<.01) and debugging time (r=.84, p<.05).

Artifact Type	Breakdowns		Errors		Debugging Time
	#	% of break.	#	% of 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)
Libraries	21	13.2%	19	18.6%	7.1 (6.9)
Faults	20	12.6%	-	-	-
Style-specific	18	11.3%	10	9.8%	3.6 (4.2)
Errors	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 (-)
Code Specification	2	1.3%	-	-	-
Failures	0	0%	-	-	-

Table 5. Frequency and percent of breakdowns and errors by artifact, and debugging time for errors.

Problem	Action	Errors		Debugging Time
		#	% of errors	Mean (SD) in minutes
Attentional Problem	Implementing	3	2.9%	5.2 (4.3)
	Modifing	4	3.9%	4.6 (7.1)
	Reusing	2	2.0%	1.2 (1.2)
	Total	9	8.8%	4.0 (5.1)
Knowledge Problem	Implementing	4	3.9%	4.2 (4.8)
	Modifying	4	3.9%	5.4 (4.0)
	Reusing	1	1.0%	5.0 (-)
	Understand	3	2.9%	6.8 (5.7)
	Total	12	11.8%	5.3 (4.2)
Strategic Problem	Implementing	11	10.8%	4.2 (3.4)
	Modifying	13	12.7%	4.7 (5.1)
	Reusing	6	5.9%	6.6 (9.3)
	Total	33	32.4%	5.1 (5.4)

Table 6. Errors and debugging time by cognitive problem and action. Only actions causing errors are shown.

ID	Prog. Time	Debugging Time		Errors	Breakdowns	Chains	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)

Table 7. Programming and debugging time, and errors, breakdowns, chains, and chain length by programmer.

5.3 Experiment Discussion

The majority of errors in these studies were (1) knowledge and attentional problems understanding implementation artifacts and (2) attentional and strategic problems implementing and modifying algorithms, language constructs, and uses of libraries. These errors forced programmers to spend nearly 50% of their time debugging on average, and caused 29 knowledge and attentional breakdowns determining faults and errors, leading to further errors. This suggests that, at least in the tasks observed in this study, even a small number of debugging breakdowns lead to significant time costs. It also suggests a likely reason for the cost: because Alice provides few facilities for inspecting the execution state of programs, programmers were unable to attain knowledge about failures, which led to knowledge and attentional breakdowns in determining faults and errors, leading to further errors. This data suggests that Alice needs better support for inspecting the state of execution and run-time interactions between program elements.

Another interesting pattern was evident in comparing P3 and P4, who both finished the task, but had vastly different strategies. For example, in creating a "while condition is true" event, P3 asked himself "How would I do this in Java?" while P4 asked the experimenter, "Just to be clear, the 'begin' part of the while event only executes once, right?" In these examples, P4 was obtaining knowledge about event concurrency, preventing insufficient specification and implementation knowledge breakdowns. Not only did P3 lack the knowledge to prevent these breakdowns, but also his experience with Java caused interfering knowledge problems, leading to strategic breakdowns and errors. These observations show that some strategies of acquiring knowledge about an unfamiliar programming system are error-prone, while others are protective.

Although the analyses in this paper limit the conclusions we can draw, they demonstrate how our model of programming errors is helpful in forming hypotheses about errors programming systems and for designing better environments. Furthermore, the data we gathered in these observations is far from limited. Future analyses inspecting programmers' specific errors will reveal more specific design guidelines for more helpful programming and debugging tools. Such analyses will also provide more insight into precisely what aspects of event-based programming make it difficult.

6. Discussion

We believe our model of programming errors supports theoretical, educational and design research by helping to describe, predict, and explain programming errors.

6.1 Supporting Reasoning

The model supports theoretical reasoning in a number of ways. First, it provides a vocabulary for reasoning about programming errors and their causes, much like Green's Cognitive Dimensions of Notations [11] supports reasoning about dimensions of programming languages. Like Green's contribution, our model makes aspects of programming errors explicit. Future studies could identify relationships between dimensions of notations and the causes of programming errors. For example, 24% of breakdowns in our study were modification breakdowns. This suggests that programming systems with structured-editing environments that have high resistance to local changes (which Green would call "viscous") may be particularly prone to modification breakdowns and errors.

Our model also supports reasoning about programming and debugging models. For example, von Mayrhauser and Vans' Integrated Comprehension Model [17] lacks any mention of breakdowns in forming mental models of specifications or code. Identifying areas where specification breakdowns can occur may help future studies of program comprehension explicitly link aspects of the comprehension process to specific error types. Our model could augment Blackwell's Attention Investment model of programming activity [2], describing how breakdowns and errors influence programmers' perception of cost, risk, and investment. Our model also supports models of debugging, such as Davies' [9]. 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 may affect debugging.

Our model also supports logical reasoning about the errors within and between environments, languages, tasks, and expertise. The studies reported in this paper are a small example of how the model is used to reason about errors within an environment, helping identify the most common breakdowns and error prone artifacts. Future studies can compare different programming systems' abilities to prevent breakdowns, which would allow statements such as "language A is more prone to strategic problems reusing data structures than language B."

Finally, the model makes explicit what can prevent breakdowns. Software engineering focuses on preventing unforeseen strategic problems in understanding, creating and modifying specifications. Programming systems focus on preventing implementation and debugging breakdowns with support such as online documentation, and colored syntax highlighting. Education focuses on avoiding knowledge breakdowns. In fact, teaching this model of errors to programmers might even prevent some breakdowns, by strengthening knowledge and providing foresight about programming and debugging strategies.

6.2 Supporting Design

The model helps design programming systems by helping to identify the breakdowns that cause specific errors. For example, from the small number of observations presented in this paper, the authors learned two important lessons about Alice: (1) on average, programmers spent 50% of their time debugging errors that were caused by unforeseen strategic problems, and (2) debugging was aggravated by knowledge and attentional breakdowns in determining faults. This suggests that a visualization of concurrent threads of execution may prevent debugging breakdowns by showing information that helps programmers better perceive and understand failures.

Also, in reference to her difficulties in modifying a complex Boolean logic statement (the breakdown chain in Figure 3), P2 remarked, "I'm really having trouble reading this…I think it's right, but I can't really tell…" This suggests that a more readable and less viscous interface for creating logical statements may prevent perception and modification errors.

Using this model of programming errors to analyze languages, environments and documentation standards could also suggest better design guidelines for tools and notations. For example, an analysis of errors in C++ would likely support the belief that operator overloading can cause attentional problems understanding language constructs. An analysis of textual specifications would likely reveal they cause unforeseen strategic problems later in development, supporting the value of runtime views in UML notation. Studying the use of print statements, breakpoints, and watches might reveal that these techniques are helpful in determining faults, but are prone to a variety of debugging breakdowns.

7. Conclusion

This paper presents a model of programming errors derived from past classifications of error types and studies of programming. We believe the model will be valuable for future research on programming errors because it provides a common vocabulary for reasoning about programming errors, while supporting the description, prediction, and explanation of programmers' errors.

8. References

[1] J. R. Anderson and R. Jeffries, "Novice LISP Errors: Undetected Losses of Information from Working Memory," *Human-Computer Interaction*, 1, pp. 107-131, 1985.

[2] A. Blackwell, "First Steps in Programming: A Rationale for Attention Investment Models," at IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, pp. 2-10, 2002.

[3] M. Burnett, et al., "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," *Journal of Functional Programming*, 11, 2, pp. 155-206, 2001.

[4] M. Conway, et al., "Alice: Lessons Learned from Building a 3D System For Novices," at Proceedings of CHI 2000, The Hague, The Netherlands, pp. 486-493, 2000.

[5] 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, pp. 61-83, 1999.

[6] S. P. Davies, "Knowledge Restructuring and the Acquisition of Programming Expertise," *International Journal of Human-Computer Studies*, 40, pp. 703-726, 1994.

[7] M. Eisenberg and H. A. Peelle, "APL Learning Bugs," at APL Conference, pp., 1983.

[8] M. Eisenstadt, "Tales of Debugging from the Front Lines," at Empirical Studies of Programmers, 5th Workshop, Palo Alto, CA, pp. 86-112, 1993.

[9] D. J. Gilmore, "Models of Debugging," *Acta Psychologica*, pp. 151-173, 1992.

[10] J. D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, 7, pp. 151-182, 1975.

[11] 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.

[12] T. R. G. Green, et al., "Parsing-gnisrap: A Model of Device Use," at Empirical Studies of Programmers: 2nd Workshop, pp., 1987.

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

[14] W. L. Johnson, et al., "Bug Catalogue: I," Yale University, Boston, MA, Technical Report 286, 1983.

[15] D. Knuth, "The Errors of TeX," *Software: Practice and Experience*, 19, 7, pp. 607-685, 1989.

[16] A. J. Ko and B. Uttl, "Individual Differences in Program Comprehension Strategies in an Unfamiliar Programming System," at International Workshop on Program Comprehension, Portland, OR, pp. (to appear), 2003.

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

[18] R. Panko, "What We Know About Spreadsheet Errors," *Journal of End User Computing*, pp. 302-312, 1998.

[19] D. N. Perkins and F. Martin, "Fragile Knowledge and Neglected Strategies in Novice Programmers," at Empirical Studies of Programmers, 1st Workshop, Washington, DC, pp. 213-229, 1986.

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

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