# A Study of Student Strategies for the Corrective Maintenance of Concurrent Software*

Scott D. Fleming†
sdf@cse.msu.edu

Eileen Kraemer†‡
eileen@cs.uga.edu

R. E. K. Stirewalt†
stire@cse.msu.edu

Shaohua Xie‡
shaohua@cs.uga.edu

Laura K. Dillon†
ldillon@cse.msu.edu

†Dept. of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA 48824

‡Department of Computer Science
University of Georgia
Athens, Georgia, USA 30602-7404

## ABSTRACT

Graduates of computer science degree programs are increasingly being asked to maintain large, multi-threaded software systems; however, the maintenance of such systems is typically not well-covered by software engineering texts or curricula. We conducted a think-aloud study with 15 students in a graduate-level computer science class to discover the strategies that students apply, and to what effect, in performing corrective maintenance on concurrent software. We collected think-aloud and action protocols, and annotated the protocols for a number of behavioral attributes and maintenance strategies. We divided the protocols into groups based on the success of the participant in both diagnosing and correcting the failure. We evaluated these groups for statistically significant differences in these attributes and strategies.

In this paper, we report a number of interesting observations that came from this study. All participants performed diagnostic executions of the program to aid program comprehension; however, the participants that used this as their predominant strategy for diagnosing the fault were all unsuccessful. Among the participants that successfully diagnosed the fault and displayed high confidence in their diagnosis, we found two commonalities. They all recognized that the fault involved the violation of a concurrent-programming idiom. And, they all constructed detailed behavioral models (similar to UML sequence diagrams) of execution scenarios. We present detailed analyses to explain the attributes that correlated with success or lack of success. Based on these analyses, we make recommendations for improving software engineering curriculums by better training students how to apply these strategies effectively.

*This material is based in part upon work supported by LogicBlox Inc and the National Science Foundation under Grant Numbers CCF-0702667 and IIS-0308063. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of LogicBlox Inc or the National Science Foundation. Licenses for Camtasia provided by TechSmith Inc.


*ICSE'08,* May 10–18, 2008, Leipzig, Germany.


## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

## General Terms

Experimentation

## Keywords

think-aloud method, concurrent programming, software maintenance

## 1. INTRODUCTION

Although the problems inherent in the design of multi-threaded software are well-known (e.g., [3, 17]), issues that arise during the *maintenance* of these systems have received less attention. Coverage of the topic in popular software engineering texts and curricula is sparse, if dealt with at all. That said, CS graduates are increasingly being asked to maintain large multi-threaded software systems. As part of the Copse project [1] at Michigan State University, we are applying rigorous empirical methods to investigate which tools, notations, and *task models* [8] best support maintainers engaged in such activities. We recently undertook an empirical study that aims to understand the strategies commonly used by CS graduates to diagnose and then correct design faults related to synchronization. This paper reports the findings of this study, including a discussion of the strategies that we observed and a collection of behavioral attributes that are more frequently observed among those participants who were successful in both finding and correcting a synchronization-related design fault than among those who were not successful. We also report a variety of observations that could inform instructors in teaching students more effective strategies for maintaining multi-threaded software systems and in employing more effective methods in conveying those concepts to students.

We collected the data in our study by observing students who were enrolled in a graduate course on formal methods as they engaged in the corrective maintenance of a system with a realistically complex multi-threaded architecture. Prior to the study, we seeded the system with a fault that caused the system to fail intermittently. Their maintenance task was to diagnose and correct the failure. The students were near in age and intellectual maturity to our target population—that is, students who have just graduated with a BS in Computer Science. We collected the data using the *think-aloud*

*method*, which is a rigorous empirical method used to model the cognitive processes that a participant engages in when performing a task [10, 28]. Long term, we intend to use these data to form formal task models [8] and then to evaluate the models for their ability to guide maintainers in successfully modifying and extending concurrent software. As a first step, this paper describes the different activities we observed and relates them to success on task.

Our study revealed two predominant strategies for diagnosing and correcting the failure. Both strategies emphasize the production of a *failure trace*, which demonstrates how the system transits among various internal states, at least one of which is a clear error state, up to the point of failure. *Execution-based tracing* aims to produce this trace by instrumenting the program and executing until such time as the program fails. In contrast, *failure-trace modeling* involves modeling what a failing trace might look like by reasoning about potential thread interleavings based on some degree of program understanding.

Although analysis of the think-aloud data is ongoing, two facts stand out. First, a significant difference in the use of failure-trace modeling was observed: those who were successful in finding and fixing the fault used the strategy more frequently than those who were unsuccessful. Second, we compared those who successfully diagnosed the fault with high confidence to those who either did not diagnose the fault correctly or did so with low confidence. Here, we observed a significant difference in frequency of what we call *strongly-articulated modeling*—that is, the articulation of a model that is "sufficiently clear and distinct" (Section 2.4). Those who correctly diagnosed the fault with high confidence performed strongly-articulated modeling more frequently.

High confidence is important because failures due to synchronization-related faults may manifest under relatively rare thread schedules, making them very difficult to reproduce. In such cases, a programmer cannot rely on execution to verify the feasibility of a candidate failure trace or testing to verify the correctness of a proposed fix. These problems have been the subject of much research in testing and debugging, with the result that several debugging tools now support replay of executions [7, 22]. Various testing approaches look to improve confidence by covering different sequences of critical events [4, 6] or by covering a reachability graph [14]. These testing approaches and replay debuggers have met with some success, but suffer problems of scale and have not seen widespread use in practice.

In our study, no participant was able to produce a suitably informative failure trace by executing an instrumented program. Rather, they produced these traces by modeling them, and a correlation was found to exist between modeling with high confidence and success. An open question is whether additional training and automated assistance in the production of trace models (failing or otherwise) could help those who were unsuccessful become more successful. For instance, some of our participants expressed difficulties producing a sufficiently articulate trace model. Such difficulties might be ameliorated by better tool support.

Another interesting observation concerns the use of execution-based tracing among those who demonstrated a capacity for strongly-articulated modeling. Almost all of the participants began by using the execution-based tracing strategy. However, those successful in diagnosing and then fixing the fault tended to use the strategy only to localize the fault. They switched to failure-trace modeling to actually produce a failure trace. Some participants continued to run an instrumented program in the background throughout the session, and many used the approach in an attempt to confirm (by absence of failure) the correctness of their solution.

We believe these data reinforce the importance of behavioral modeling as a skill in the CS graduate's repertoire. Our experience suggests that concurrent-software maintenance tasks provide a wealth of opportunities for developing these skills. Some of the students in our study appear to have performed no modeling whatsoever. All of these students continued to invest in the execution-based tracing strategy long past the point of diminishing returns and ultimately were unsuccessful in diagnosing the fault. Perhaps, we could develop greater "buy-in" regarding the limitations of this strategy through laboratory exercises using artifacts similar to those used in our study. Here, students would gain first-hand experience with these limitations. In addition, we believe a modeling tool that facilitates the construction of incomplete sequence diagrams to quickly represent and improve models along the quality spectrum may both aid in the development of this skill and be useful in practice.

The remainder of this paper is structured as follows. Section 2 provides background on the think-aloud method and concepts used in describing to our study and our findings. We describe the materials used and the procedure followed to collect our data (Section 3) followed by the data (Section 4) and a statistical analysis (Section 5). Finally, we provide discussion of the study and future work (Section 6) and wrap up with conclusions (Section 7).

# 2. BACKGROUND

## 2.1 Think-aloud method and user studies

The *think-aloud method* is a rigorous empirical technique used to obtain a model of the cognitive processes that take place during an activity or to test the validity of a proposed model [10, 28]. A *think-aloud protocol* is a transcript of the verbalizations of the participants as they are engaged in some activity. An associated *action protocol* describes what participants do as they engage in the activity. These transcripts are later analyzed by segmenting them and mapping their components into the concepts and operations of a given candidate task model. This mapping is produced by means of a *coding scheme*, which specifies how patterns of speech and actions are expected to correspond to the performance of specific tasks in the candidate model. Domain experts develop these coding schemes, and independent analysts, called *coders*, apply them to raw protocols to yield *encoded protocols*. The fitness of a candidate task model can be assessed by evaluating how well it "covers" a set of protocols—that is, by how well the model matches the observed behaviors.

As in Newell and Simon's study [19], we intend to incrementally refine our encoding scheme to derive a predictive model that fits the data. Then, we will evaluate the models based on newly collected sets of such protocols. In our initial analysis of the protocols collected in this study we devised some simple coding schemes, which were applied by members of the research team. We will further refine the coding schemes, and multiple independent coders will re-annotate the videos.

The time and effort required from both participants and researchers for such intensive user studies is substantial, and the number of participants typically ranges from 5 up to 20. For example, Soloway and colleagues [26] used 20 participants in their think aloud study of maintenance tasks; Wallace and colleagues [30] used 5 participants in their think-aloud study of end-user programming; Ko and colleagues [12] used 10 participants in their study of how programmers use interactive development environments for performing maintenance tasks; and LaToza and colleagues [15] used 13 participants in a study to understand how programmers think about design when performing maintenance activities.

Studies of programmers performing software maintenance tasks serve a variety of purposes. For example, Robillard and colleagues [21], and LaToza and colleagues [15] looked for differences in performance between novices and experts and for differences in the procedures they use; Vans and colleagues [29] validated a program comprehension model; Prechelt and colleagues [20] evaluated the effects of design patterns on maintenance; and Ko and colleagues [12] determined how experts use an interactive development environment to support maintenance. This latter study is closest to ours in goals and methods: "The goal of our study was to discover fundamental activities in maintenance work in order to inspire new ideas for more helpful tools" [12]. They analyzed screen-capture videos of 10 programmers working alone on the same maintenance tasks and found that participants interleaved three basic activities: (1) identifying a small set of code fragments pertinent to the maintenance task; (2) navigating the dependencies between these code fragments; and (3) repairing code fragments. As a result of this analysis, Ko and colleagues proposed a number of novel ideas for how a maintenance-oriented IDE could better support maintenance activities. The conclusions of the Ko study are subject to many of the same threats to validity as our study due to the similarities in methods and size, except that participants were not asked to think aloud in the Ko study. However, the presence of think-aloud data in our study permitted us to ascribe purposes to many of the activities that participants performed, and to determine what participants learned from the activities and what misconceptions they harbored.

None of the studies mentioned above specifically studied how programmers correct synchronization flaws. The applications modified by participants in the studies by Robillard and colleagues [21] and by Ko and colleagues [12] were most likely multi-threaded; however, the maintenance tasks involved functional modifications of features for which synchronization issues, if any, would have been incidental.

## 2.2 eBizSim

We developed a software suite to use for this study. The *eBizSim suite* simulates an e-business server that accepts and processes requests from remote clients. The suite comprises a multi-threaded *server* program and a separate *stress tester* program, for use in load testing the server. The server accepts network connections from remote clients, receives requests from the clients over these connections, and simulates processing of the requests. The server comprises multiple threads, each of which plays one of two distinct roles—that of a *listener* or that of a *handler*. A lone listener thread accepts client connections and places the requests received over these connections on a shared queue. Meanwhile, multiple handler threads contend for requests by synchronizing on the shared queue. The handlers themselves are organized as a thread pool. Loosely based on Schmidt and colleagues' *reactor pattern* [24], this thread-pool architecture mimics a realistically complex multi-threaded server. However, because the server only simulates the processing of client requests, its size is manageable enough for a user study in which the participants are seeing the code for the first time.

We seeded the eBizSim server with a design fault related to the proper use of condition synchronization in the transfer of requests between the listener and handler threads. This fault manifests in a failure under certain timing and load constraints. Using the stress tester, which provides a GUI dial for adjusting the speed at which requests are sent to the server, it is often possible to reproduce the failure within a time frame of two to five minutes. However, we have never been able to reproduce the failure once the server runs without fail for more than five minutes. Thus, in practice, it is often necessary to restart the server multiple times to produce the failure. We chose this design fault to be representative of the class of synchronization-related faults that are difficult to reliably reproduce by running the program.

The fault stems from the way the listener and handler threads synchronize as they access the queue of connection requests. Henceforth, we refer to this queue as the *request queue*; its elements are instances of a class `Request` whose internal structure is not salient to this discussion. The request queue is managed by an object called the *pool*, which encapsulates and provides synchronized access to both the request queue and the thread pool. The pool defines a mutex lock for each resource that it manages and a host of queue/pool-specific operations, each of which is implemented so as to acquire (and release) the appropriate mutex at the beginning (and the end) of the operation. We concentrate on the operations that manipulate the request queue, as this code contained the seeded fault.

The request queue is accessed through the operations `submit_request` and `retrieve_request`. Figure 1 depicts the implementations of these operations. Both methods acquire and release a mutex lock called `queue_lock_`. Thus, all calls to `submit_request` and `retrieve_request` execute under mutual exclusion. Moreover, calls to `retrieve_request` may block when the request queue is empty. The conditional blocking logic is implemented in lines 16–19 in `retrieve_request`, and the corresponding signaling logic (used to resume blocked threads when the blocking condition may have changed) is implemented in lines 6–9 in `submit_request`. The variable `nonempty_queue_cond_` refers to a *condition variable*, upon which threads may issue the operations `wait`, `signal`, and `broadcast`.[1] The variable `queue_waiters_` records a count of the number of handler threads currently waiting for a request to be placed in the queue. The `submit_request` method checks the value of this counter to decide whether it needs to signal the condition variable.

The fault, which we seeded, appears on line 16, where we replaced the line:

```
while(request_queue_.empty()) {
```

with the line:

```
if(request_queue_.empty()) {
```

To see how this fault may manifest in a failure requires reasoning about possible interactions between three threads—two handler threads and the listener thread—during concurrent activations of `retrieve_request` and an activation of `submit_request` when the request queue is empty.

## 2.3 Strategies for producing failure traces

As mentioned previously, our study revealed two predominant strategies for producing concrete failure traces from which to diagnose the fault. The execution-based tracing strategy aims to produce the trace by instrumenting the program and then executing the program until a failure occurs. Because this strategy should be familiar to most readers, we will not define it further.

The other strategy (i.e., failure-trace modeling) aims to deduce a failure trace as follows. The analyst first formulates a candidate sub-trace that ends in a clear error state. We refer to this sub-trace

[1] Our implementation is in C++ and uses primitives from the ACE toolkit. Readers familiar with Java can think of `signal` as analogous to `notify` and `broadcast` as analogous to `notifyAll`.

```
1  void Pool::submit_request(Request* request)
2  {
3    queue_lock_.acquire();
4    request_queue_.push_back(request);
5
6    if (queue_waiters_) {
7      nonempty_queue_cond_.signal();
8      --queue_waiters_;
9    }
10
11   queue_lock_.release();
12 }
```

```
12 Request* Pool::retrieve_request()
13 {
14   queue_lock_.acquire();
15
16   if (request_queue_.empty()) {
17     ++queue_waiters_;
18     nonempty_queue_cond_.wait();
19   }
20   if (request_queue_.empty()) {
21     queue_lock_.release();
22     return 0;
23   }
24
25   Request* request = request_queue_.front();
26   request_queue_.pop_front();
27
28   queue_lock_.release();
29   return request;
30 }
```

**Figure 1: Key synchronization methods in the eBizSim server.**

as an *error suffix*. She then attempts to verify that the error suffix is *feasible*, meaning that it is consistent with an actual execution trace. Both of these activities rely on program comprehension; however, they exhibit a stark difference in complexity. Candidate error-suffix formulation is relatively lightweight but also prone to yielding spurious errors—that is, behaviors that could not actually arise in an execution of the program. By contrast, feasibility analysis can be extremely difficult and time consuming. This difference in complexity is similar to that observed of race detection in debugging, where detecting *apparent races* requires local reasoning and is thus generally efficient, whereas detecting *feasible races* requires nonlocal reasoning and is, in general, NP-hard [18]. The general process and the models produced are best illustrated by example. We use the UML 2.0 sequence diagram, with some adornments, to depict these models.

Figure 2 depicts an (infeasible) candidate error suffix, as articulated by one of the participants in our study. Here, two handler threads (denoted $h_1$ and $h_2$) concurrently attempt to retrieve a request from an empty request queue. Because the queue is empty, both threads wait on `nonempty_queue_cond_` (abbreviated `neqc` in the figure). Shortly thereafter, the listener thread (denoted $l$) invokes a `submit_request` operation, which adds a request (denoted `r`) to the queue and, according to the figure, invokes a `broadcast` operation on condition variable `neqc`. In response to the broadcast, both handler threads resume and attempt to reacquire `queue_lock_`. Here, $h_1$ acquires the lock and is able to proceed, after which it pulls `r` off of the queue and releases the lock. Thread $h_2$ acquires the lock next and and proceeds. But because the queue is once again empty, the `retrieve_request` method returns 0, which causes $h_2$ to enter an error state.

A quick word on notational conventions: To distinguish type/class names (e.g., `Pool`) from role names (e.g., *Handler*), we render the latter in italics. To indicate that the queue is empty at the start of the suffix and show how it mutates during the trace, the lifeline of the pool object (denoted `rhp`) is adorned with *object-state predicates*—for example, $q == [\,]$, which are depicted inside dashed roundtangles connected to the object's lifeline by a dashed horizontal line. Similarly, *activation-state predicates*, which are depicted inside roundtangles that are centered atop corresponding activation bars, show when the thread executing the activation

- acquires or releases the lock on the queue (the assertion `qLocked` becomes true or false respectively), and
- blocks on or awakens from waiting on `nonempty_queue_cond_` (the assertion `waitg(neqc)` becomes true or false respectively).

To indicate that the waiting threads are awakened by a broadcast, as opposed to a signal, we position the activation-state predicates so as to depict the awakening at the same instant of the pool's lifeline. Additionally, to reduce nonessential clutter, messages corresponding to method invocations are rendered abstractly, without showing all the objects involved, but showing the thread that invokes them. For example, calls to `retrieve_request` are actually made from within activations of another pool method, `dispatch_request` (omitted for clarity), and a more detailed model would show the `retrieve_request` messages emanating from activations (of `dispatch_request`) that are attached to the pool's lifeline; as the figure does not depict activations of `dispatch_request`, we instead show the messages emanating from the lifelines of the thread executing the elided activations.

An analyst might construct such a model to use in determining if the failure is a result of a call to `retrieve_request` that returns null. The model documents the initial state of the relevant object (i.e., the pool), and it describes the number and role(s) of the interacting threads. Additionally, the activation predicates are consistent with the semantics of available synchronization primitives (in this case, with `acquire`, `release`, `wait` and `broadcast`). Such a model represents a candidate error suffix. If the initial state of the model is consistent with a reachable program state, and if the actions and state transitions in the model are consistent with the code, then the model describes a feasible trace that ends in an error and that therefore exhibits a fault.

A candidate error suffix exhibits an actual fault only if it is feasible. Such a model seldom begins in the initial state of the program, and it usually elides many details. Thus, when reasoning about thread interactions, the programmer can easily construct a model that is not feasible. This is especially true in a maintenance context where the programmer may lack a global view of the program. The model in Figure 2 is not feasible, because the listener thread uses `signal` rather than `broadcast` to notify a handler thread of a request in the queue. Thus, only one of the handlers will be awakened during the activation of `submit_request`, whereas the model depicts both as being awakened.

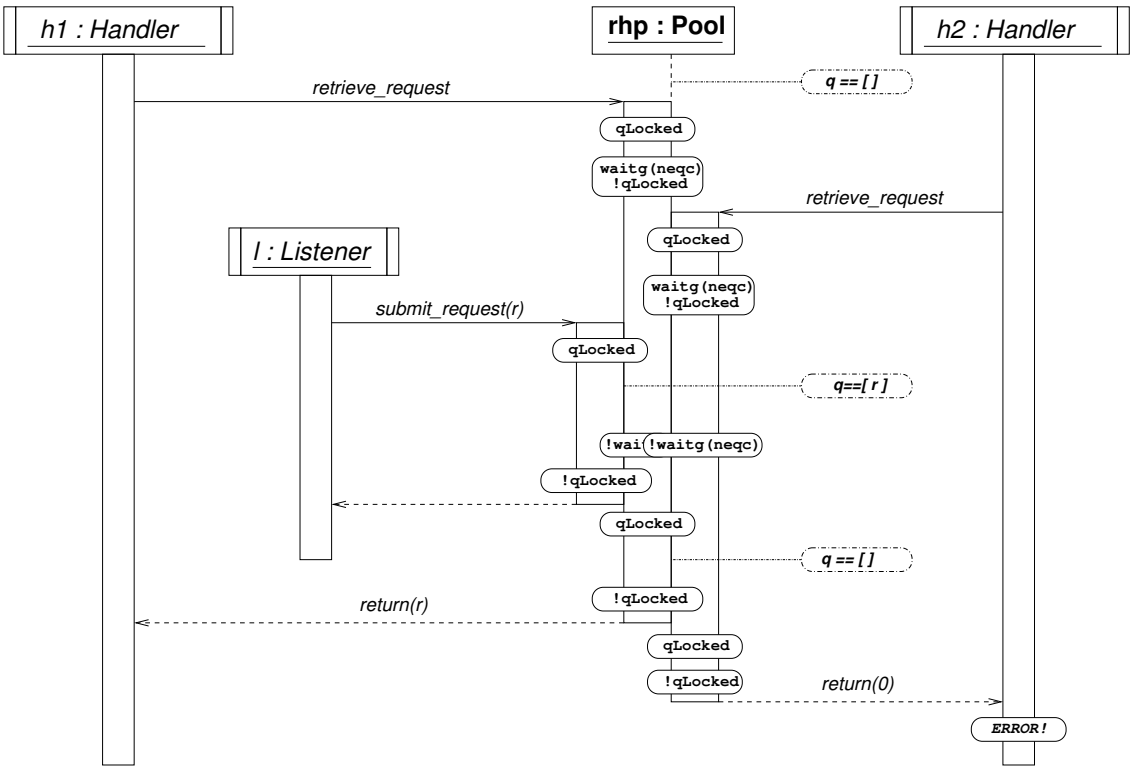


**Figure 2: Strongly-articulated, yet infeasible model of an error trace.**

Generally speaking, a deep analysis is required to decide whether a candidate error suffix is feasible. Given an error suffix, this analysis can be performed by:

1. instantiating its abstract components with concrete objects, activations, threads, and states—that is,
   - binding the abstract actors (depicted as active objects in the figure) to activations of operations on concrete objects by concrete threads, and
   - verifying that the activation- and object-state transitions depicted in the error suffix are consistent with the implementation of the associated operations
2. checking that the initial state of the instantiated model is consistent with a state reachable from the initial state of the program.

As suggested previously, this task may involve global reasoning and is generally very difficult. A key goal of our work is to learn how successful programmers cope with this complexity.

## 2.4 Strength of model articulation

In studying the failure-trace modeling strategy, we quickly discovered vast differences in the quality (i.e., clarity and distinctness) of the models articulated by our participants. Some referred to threads only in the abstract, whereas others also identified which thread acted as the listener and which threads acted as handlers. Additionally, some participants articulated the interactions that might culminate in an error with sufficient detail that we could construct an error suffix such as is depicted in Figure 2, whereas others did not articulate necessary information for formulating an error suffix. For example, a participant might say that an operation is invoked, but not say which thread invokes the operation; or he might not say when a thread acquires (releases) a lock or when it waits (awakens from a wait).

We classify as *strongly articulated* a model that:

- describes a collaboration among distinct actors and objects with well-defined synchronization states,
- describes how actions by the actors cause the objects and other actors to transition among these synchronization states, and
- is articulated (i.e., drawn, verbalized, etc.) in sufficient detail to enable formulation of a well-formed UML sequence diagram, such as the one in Figure 2.

In our study, we observed a significant prevalence of strongly articulated modeling among those participants who diagnosed the problem with high confidence (Section 5.1). This result is interesting because high confidence in fault diagnosis indicates a strong belief that the error being modeled is not spurious, and as mentioned previously, error-suffix formulation is prone to the production of spurious error traces. The prevalence of strongly-articulated modeling in the group that successfully diagnosed the problem with high confidence suggests that these participants may have used strongly-articulated modeling as a means of coping with the complexity of feasibility analysis.

# 3. PROCEDURE

Materials used in the study included written pretests and posttests, and a computer workstation. The workstation was

Bug Report / Change Request

We are experiencing a problem with the eBizSim server program, wherein it intermittently exits with the error message:

```
error: Pool::dispatch_request() failed
```

This error has been fairly difficult to reproduce. So far, the most reliable way we have found to reproduce it is to run the stress tester with a setting of 4.27. Even with this setting, the program may take several minutes to exhibit the error. Occasionally, the program will run at the above setting for a long time (on the order of 5 minutes) without failing. In these cases, restarting the server and the stress tester seems to help in drawing out the error.

**Figure 3: Bug report provided to participants.**

equipped with standard Web browsing software (e.g., Internet Explorer and Firefox) and programs to permit remote access to the machine on which the eBizSim software and editing tools familiar to the participant were installed. In addition, the workstation was outfitted with the "Camtasia" video capture software and microphone headsets, which provided the ability to capture the screen interactions and speech of the participants in the form of a video of the computer screen, with voice-over from the participant.

Fifteen students in a graduate-level formal methods course at Michigan State University participated in the study. All students registered for the course happened to be male. The participants received group instruction, in the form of a 50 minute lecture, on concurrency constructs and their implementation using the ACE wrappers toolkit [25]. The goal of this lecture was to ensure that participants were well-prepared to undertake the assigned maintenance task and to mitigate the effects of differences in prior knowledge on their performance. The participants then took a pretest on concurrency terminology and concepts.

Participants were scheduled for individual 3-hour sessions, conducted in a private office. During the first 15-20 minutes, participants were introduced to the equipment and environment, audio collection was calibrated, and participants engaged in think-aloud on a trial task, correcting a fault in the implementation of a bubblesort procedure. "Prompters" trained in the think-aloud method accompanied the participants as they engaged in the maintenance task and, as needed, prompted the participants to verbalize their thought processes. Participants were then given a brief tour of the directories containing the eBizSim software and provided a bug report. Figure 3 depicts the bug report, which describes the output associated with the failure as well as some tips on how to reproduce the failure. Scratch paper, a brief guide to concurrency constructs in ACE, and a C++ manual were also provided. Participants were permitted to browse the Internet as they deemed necessary.

Participants were allotted up to 150 minutes to complete the task. Those who completed the task sooner could stop the session at that time. Following the sessions, participants took a posttest designed to evaluate their comprehension of the software system.

# 4. RESULTS

We developed a collection of Boolean behavioral attributes to use in assessing and comparing the performance of each participant and the strategies they used in solving the problem. We now describe these attributes and the criteria we used to assess them.

We measure success in diagnosing the fault using three attributes:

**loc:** identified that what triggered the failure was the return of 0 on line 22 in `retrieve_request` (see Figure 1).

**pat:** recognized violation of the wait-in-while pattern.

**rat:** provided rationale to explain why the faulty code is faulty.

We record loc for each participant who was able to narrow his search to the code segment that contains the seeded fault. This attribute is recorded without regard to whether the participant could explain why the code was faulty or whether the participant was ultimately able to correct the fault. We record rat for each participant who was able to explain why the faulty code is in error. As evidence of rat, we look for verbalizations, such as: "when `wait` returns, the queue is not guaranteed to be non-empty," or, "I now see how the queue can be empty when the thread exits the if block." Clearly, rat implies loc, but the converse is not true.

The pat attribute indicates whether the participant recognizes a violation of the "wait-in-while" pattern, which is used to implement condition synchronization in monitors. This pattern is discussed with prominence in texts on concurrency (e.g., [17]), and misuses (e.g., guarding the `wait` with an `if` as opposed to placing it inside a loop) are often used in homework assignments and on exams. The lecture that we gave prior to administering the pretest explicitly discussed the pattern and why it is necessary. Prior work suggests that students often use a pattern-based strategy for designing condition-synchronization logic and have difficulty reasoning about novel uses [13]. We collected the pat attribute to judge whether students used a similarly shallow strategy in corrective-maintenance tasks.

We measure success in correcting the fault using two attributes:

**fix:** replaced the faulty `if` statement with a `while` statement in `retrieve_request`.

**noNew:** fixed the fault and introduced no new faults.

We record fix for each participant who successfully fixed the seeded fault, which essentially means that he changed the `if` block into a `while` block in the `retrieve_request` method. We record noNew for each participant who fixed the seeded fault as indicated and did not introduce a new fault.

We look for evidence of the trace-modeling strategy using four attributes, which indicate various levels of modeling:

**mod:** participant gave evidence of modeling some interaction among multiple threads synchronizing over shared data.

**err:** participant gave evidence of modeling an interaction that manifests in a clear error state.

**art:** participant produced a model that was strongly articulated.

**succ:** participant produced a strongly-articulated model that accurately describes the error that led to failure.

We count as evidence of modeling any description (be it a verbalization or a drawing) that supposes the existence of two or more threads and one or more shared objects and that proceeds to develop a sequence of actions by and among these entities. Clearly, err, art, and succ imply mod. However, a modeled interaction need not involve an error state (or indeed have anything to do with the failure in question), and modeling may or may not be strongly articulated. Moreover, err does not imply art, and art does not

imply err. Additionally, succ implies but is not equivalent to err $\wedge$ art.

We look for evidence of the execution-based tracing strategy using three attributes, which record various levels of this activity:

inst: instrumented the code by adding diagnostic print statements.

refine: refined diagnostic statements to expose more detailed internal-state information based on the diagnostics provided by a previous failing run of the program.

found: produced a failing trace whose diagnostics indicate the error state that led to failure and the sequence of synchronization events that caused the program to enter this state.

Clearly, refine and found imply inst. We collected refine to measure the level of investment in this strategy, that is, was the participant taking an iterative approach to narrowing down the source of the fault? The criteria for recording found are quite strong—a run has to produce a failure trace whose diagnostics show at least two handler threads waiting, the listener thread arriving and signaling both of them, one handler waking and emptying the queue, and the second waking to an empty queue.

We also observed some participants appealing to what seemed to be little more than luck:

luck: tweaked the synchronization code without explanation or expectation of success, seemingly in an attempt to luck into a fix.

We record luck when a participant modifies the synchronization logic or other logic that could affect thread schedules and then executes the program to "see what happens." Examples include reordering lines 7 and 8 in `submit_request`, commenting out the return statement on line 22 in `retrieve_request`, and removing all of the code involving the `queue_waiters_` variable. In each of these cases, the participant uttered something to indicate a lack of any real expectation that the tweak would solve the problem—for example, "if that actually was the problem, I'll be upset."

Finally, the attribute conf indicates an expression of high confidence that the participant has correctly diagnosed the fault. As evidence of conf, we look explicit declarations of confidence such as "[we are] really confident that we identified the problem correctly," and exclamations that implicitly indicate confidence such as, in reference to a fault diagnosis, "that's gotta be it!" Notice that this attribute indicates a participant's opinion of his performance and need not reflect real success.

Table 1 depicts our record of each of these attributes for each of the fifteen participants.

# 5. ANALYSIS

We analyzed participant videos, pretests and posttests. Multiple researchers viewed the videos and annotated the protocols to indicate success at diagnosis, confidence in diagnosis, success at fault correction, the number of times the program was executed, the number of failed runs, and the behavioral attributes described in the previous section. We developed keys for the pretests and posttests, and scored the tests. We recorded the annotations in spreadsheets, and we analyzed the data to find significant differences between the groups that resulted based on success at diagnosis and fault correction.

## 5.1 Diagnosis with high confidence

To study how participants differed based on their success in diagnosing the fault, we divided the participants into two groups: a group of participants who successfully diagnosed the fault with high confidence, and a group of participants who either did not diagnose the fault or who did so with low confidence. We then looked for statistically significant differences between these groups.

We found no statistically significant differences in prior knowledge as measured by scores on the pretest, nor any significant difference in performance on the posttest.

We did find significant differences between the groups on attributes related to recognizing violations of common synchronization patterns (pat), modeling (mod), and tweaking of code (luck). The successful, confident group tended to recognize the violation of the wait-in-while pattern (pat) ($p < 0.001$) and to employ modeling (mod) ($p < 0.05$), and tended *not* to make "tweaking" changes to the code (luck) ($p < 0.05$).

In a more detailed look at differences between the modeling behaviors of the groups, we found that the successful, confident group tended not only to have succeeded in modeling an accurate failure scenario (succ) ($p < 0.05$), but either to have modeled a candidate error trace (err) or to have generated a strongly-articulated model (art) ($p < 0.01$), or to have done both (err $\wedge$ art) ($p < 0.001$). In fact, all of the participants who correctly diagnosed the fault with high confidence both modeled a candidate error trace and generated a strongly-articulated model (err $\wedge$ art), whereas most participants from the other group did only one or the other.

That these groups showed no significant difference in prior knowledge on the pretest, yet a marked difference in success at diagnosis, suggests that the use of modeling may be a key element in diagnosing such synchronization-based errors. Further, the data suggest that both strongly-articulated modeling and the modeling of a candidate error trace are important.

The prevalence of recognition of the violation of the wait-in-while pattern by the successful group suggests that training with such common patterns is another key element. That roughly half of the participants did not note this violation, despite a recent lecture on the topic in class, suggests that lecture alone may be insufficient to convey this concept and that hands-on exercises may be necessary. Further, we note that all participants who recognized this violation of the wait-in-while pattern did so in the context of modeling. No participant merely recognized that the pattern had been violated and fixed the code solely on that basis.

It seems likely that the correlation of the "tweaking" strategy with failure to correctly diagnose is a by-product of the failure to comprehend the nature of the fault; those participants didn't know what else to do. The negative correlation between this strategy and a successful outcome suggests that programmers should not waste time in this way, and might better spend their time attempting to localize the error, looking for violations of well-established patterns, modeling the behavior of the system, and constructing candidate error traces.

## 5.2 Fault correction

To study how participants differed based on their success in correcting the fault, we divided the participants into three groups: a group of participants who successfully corrected the fault and did not introduce any new errors (the *successful* group), a group of participants who corrected the fault but introduced a new error (the *partially successful* group), and a group of participants who did not correct the fault (the *unsuccessful* group). We then compared the successful and unsuccessful groups and looked for statistically significant differences between these two groups.

| | loc | rat | pat | fix | noNew | mod | err | art | succ | inst | refine | found | luck | conf |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **par01** | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par02** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par03** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par04** | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| **par05** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **par06** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| **par07** | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| **par08** | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| **par09** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| **par10** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| **par11** | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par12** | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par13** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **par14** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| **par15** | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

**Table 1: Observed values of attributes by participant protocol.**

We found no statistically significant differences between these two groups in prior knowledge as measured by scores on the pretest, nor did we find any significant difference in their performance on the posttest.

We found several significant differences between the groups on attributes related to failure-trace modeling and recognition of patterns of synchronization logic. The successful group tended to have recognized the violation of the wait-in-while pattern (pat) ($p < 0.05$), to have employed modeling (mod) ($p < 0.05$), to have modeled a candidate error trace (err) ($p < 0.05$), and to have generated a strongly-articulated model (art) ($p < 0.05$).

No significant difference was found between the groups in the use of execution-based tracing (inst). Both groups employed the strategy. However, those in the successful group appeared to use such tracing to localize the error to the `retrieve_request` method, and then to switch to other strategies. For example, one successful participant ran the program once to confirm that the failure was produced, and then spent some time reading the code and locating the `dispatch_request` method. He then instrumented the code in that method to determine which of the several steps in that method was failing. While the program was running, he continued to study the code, but was "hesitant to do too much" until he "narrowed things down by at least one level." Once he achieved that, he sketched out a sequence diagram. However, he continued to run the program in the background, saying, "We'll just keep this going while we think."

In contrast, those in the unsuccessful group appeared to continue to pursue an execution-based tracing strategy further. We plan a more detailed analysis of the number and timing of execution attempts and failures to further investigate the transition from one strategy to the other.

Overall, analysis of the data on fault correction leads to similar conclusions as the analysis of the data on fault diagnosis. Strongly-articulated modeling (art) and the modeling of a candidate error trace (err) are important elements of success in the correction of synchronization-related faults, as is hands-on experience with the use of common patterns of synchronization logic.

# 6. DISCUSSION AND FUTURE WORK

We recognize several threats to validity in this work. The first concerns how well our seeded fault represents the kind of synchronization faults that arise in practice. To address this concern, we will develop materials that span a larger space of synchronization-related failures and perform further think-aloud studies using these materials. We will use the classification of concurrency failures in [16] to guide our design. Other concerns include the limited scale of both the program and the change activities, and the absence of strong incentives for participants to succeed. These problems are difficult to address given the nature of a think-aloud study. Participants can be asked to participate for a relatively limited length of time. The natural incentives to succeed that exist in a real-world setting are difficult to duplicate in an experimental study involving students. Nevertheless, we found significant results related to the prevalence of modeling and the recognition of violations of common patterns of synchronization logic among those who successfully diagnosed and fixed the fault. We will develop other kinds of studies (e.g., case studies) to validate our results on programs and change activities of larger scale and with a more realistic structure of incentives.

Another potential threat to validity concerns both the size and composition of the student pool. Whereas 15 participants is on the high end for a think aloud study, it represents a small pool from which to generalize to all recent graduates with a BS in Computer Science. Moreover, students who choose to enroll in a graduate course in formal methods may not represent the general population of students who graduate with a BS in computer science. Further, the participants in this study were all male. We will address these threats in future work by repeating our study with a wider sampling of participants. In particular, we are eager to include industrial practitioners in our studies.

The use of the think-aloud method is a potential threat to validity because the cognitive resources required for introspection may affect how participants perform. Fortunately, numerous studies show that participants who only are asked to "verbalize their inner dialogue", as were the participants in this study, perform comparably on measures of performance with participants who are not asked to think aloud [9]. After an hour into our study, one participant who had thought that the method would be a hindrance stated, "[Talking aloud] turned out not to be a big deal. Especially while I was thinking through sequences of events, I pretty quickly became unaware of the fact that I was talking out loud."

We believe the findings of this study have several important implications for teaching courses in concurrency and software maintenance. First, although execution-based tracing was useful for lo-

calizing the error, none of our participants who relied solely on that approach was able to properly diagnose (rat) and fix (fix) the fault. On the other hand, every successful participant exhibited modeling at some level, and those who diagnosed the fault with high confidence exhibited strong model articulation. An interesting research question is whether better tool support for modeling might help those who were unsuccessful become successful by assisting in the strong articulation of candidate error traces and/or in feasibility analysis. An automated tool that builds sequence diagrams [5, 23, 27] could be useful in an educational setting to support trace modeling. Such an automated tool could render a scenario as a sequence diagram, which should be easier to interpret than a linear listing of events and attribute values over time. Moreover, it could relate the elements in a sequence diagram to constructs in the code to aid program comprehension. If students could generate sequence diagrams as easily as they can generate execution-based traces, they might be more likely to invest in trace modeling.

Further, that some participants (4 of the 15 in our study) spent so much time in execution-based tracing, despite expressing the belief that this strategy is unlikely to be effective, again emphasizes the importance of assignments that give students actual experience in maintenance of multi-threaded software.

Currently, most courses that deal with concurrency and synchronization discuss common synchronization idioms and patterns. The wait-in-while pattern is a classic example: Seeing a `wait` statement inside an `if` block should have raised a red flag for our participants, especially given that we had spent time in class discussing the proper use of the pattern and the dangers of violating it. Additionally, the lecture slides contained a full slide on just this pattern. Our finding that only those participants who performed modeling activities recognized the violation of the wait-in-while pattern suggests that a deeper engagement with the subject matter is required and that hands-on exercises would likely be of benefit. More generally, the relatively low success rate and extremely low confidence rate among participants suggests that the software-engineering curriculum should include more experience in the maintenance of multi-threaded software.

In summary, with regard to multi-threaded systems, our findings support teaching students that trace modeling or trace modeling in combination with execution-based tracing is likely to be more effective than execution-based tracing alone, and that programmer time is likely more fruitfully spent in such modeling activities than in code "tweaking" or in pursuit of an elusive execution-based error trace.

We expect to address questions raised in this initial analysis both by additional data collection and analysis as part of this study and also by conducting further studies. With respect to the current study, variations in the approaches taken by the participants may be attributable to differences in individual characteristics and abilities such as working memory, spatial visualization abilities, distractability, classification skills, fluid intelligence, and learning-style preference. We will administer standard tests to assess these differences during the second phase of this study.

Finally, we will continue to pursue the development of a task model for the maintenance of concurrent software by formalizing the coding scheme we used to collect the data in this paper and developing a verbalization theory, which maps verbalizations to codes, that multiple independent coders can apply to reliably encode these protocols. Once we develop a sufficient verbalization theory, we will be able to perform a more precise analysis of correlations among approaches (execution-based tracing, trace modeling, recognition of synchronization patterns) and success at diagnosis and correction of synchronization-related faults.

# 7. CONCLUSIONS

This research provides insights into the strategies commonly employed by CS graduates to diagnose and correct design faults related to synchronization. Our results suggest that

1. Programmers are primarily concerned with developing a failure trace to diagnose and correct the fault.
2. Although programmers seem to prefer to use execution-based tracing to produce a failure trace, the strategy, by itself, is ineffective.
3. Trace modeling is an effective means of producing a failure trace—especially, when the models are strongly articulated.

Although all participants used execution-based tracing, there seemed to be a limit to its usefulness. Participants that employed only execution-based tracing were all unsuccessful. Successful participants tended to use execution-based tracing to localize the fault and then use trace modeling to complete the diagnosis. It makes sense that this would be the case, because of the additional challenges concurrent programs create for execution-based tracing. Print statements must be placed judiciously so as not to affect the scheduler in a way that makes the failure more difficult to reproduce. Placing or removing statements in the code, such as `sleep`, to force certain interleavings is also very difficult to do correctly.

Strongly-articulated modeling, the modeling of a candidate error trace, and active knowledge of common patterns of synchronization logic appeared to be key elements to confidently diagnosing and fixing a synchronization-related fault. No participant succeeded using only execution-based tracing or through recognition of a violation of the wait-in-while pattern. All successful participants used modeling of some kind (mod) with most performing strongly-articulated modeling (art) and generating a candidate error trace (err).

Our findings have led us to make several recommendations. We should better educate SE students as to the limitations and applicability of the execution-based tracing strategy. We suggest designing exercises in which students perform different program comprehension-related tasks—for example, understanding control flow and possible thread interleavings—and maintenance on small programs, such as the one from our study. Such exercises would provide students hands-on experience in attempting these strategies and in experiencing the success or failure of these attempts. Our study did not provide students with access to tools such as VeriSoft [11], which provides a debugger-style interface that separates output by process and enables users to take full control of the process scheduler. Whether participants who pursued execution-based tracing exclusively might have fared better using such tools is a question for future studies.

To make students better trace modelers, we need to improve their behavioral modeling skills, especially in the context of maintaining actual concurrent programs. Too often, behavioral modeling is taught only in the context of software design. This creates two problems: students do not apply the modeling activity to actual code and they do not get experience reverse engineering models from existing programs. We see exercises that have student fulfill program comprehension tasks on small concurrent programs using behavioral modeling as an effective way to help students gain skills applying modeling to the sorts of problems that arise during software maintenance.

We also recommend training students in tools and techniques for externalizing sequence diagrams. Participants in our study tended to create the models in their minds, and we assume that this is taxing on working memory. Such strain might decrease the ability to

work effectively with a model. For example, we observed participants creating a model that represented a correct execution of the system and then analyzing that model for alternative interleavings that might lead to an error. We assume it would be easier for them to find such alternative interleavings using a diagram depicted on a piece of paper, rather than a diagram kept completely in their heads. Tools such as EclipseUML [2] provide model diagraming facilities that could help students create such external representations.

# 8. REFERENCES


[1] Copse project. http://www.cse.msu.edu/sens/copse/, 2008.

[2] EclipseUML. http://www.omondo.com/, 2007.

[3] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[4] M. Biberstein, E. Farchi, and S. Ur. Choosing among alternative pasts. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[5] L. C. Briand, Y. Labiche, and Y. Miao. Towards reverse engineering of UML sequence diagrams. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[6] J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 313–322, New York, NY, USA, 2007. ACM.

[7] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multi-threaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[8] D. Diaper, editor. *Task Analysis for Human Computer Interaction*. Ellis Horwood, 1989.

[9] K. A. Ericsson. Valid and non-reactive verbalization of thoughts during performance of tasks. *Journal of Consciousness Studies*, 10(9–10):1–19, 2003.

[10] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, 1993.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[12] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 126–135, New York, NY, USA, 2005. ACM.

[13] Y. B.-D. Kolikant. Learning concurrency: evolution of students' understanding of synchronization. *Int. J. Hum.-Comput. Stud.*, 60:243–268, 2004.

[14] P. V. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent software. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 14–23, New York, NY, USA, 1996. ACM.

[15] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 361–370, New York, NY, USA, 2007. ACM.

[16] B. Long and P. A. Strooper. A classification of concurrency failures in Java components. In *Proc. of the IEEE International Parallel and Distributed Processing Symposiom (IPDPS'03)*, page 287, 2003.

[17] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 2000.

[18] R. H. B. Netzer and B. P. Miller. What are race conditions?: Issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[19] A. Newell and H. Simon. *Human Problem Solving*. Prentice Hall, 1972.

[20] L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, 2001.

[21] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[22] M. Ronsse and K. D. Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[23] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[24] D. Schmidt et al. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John-Wiley & Sons, 2000.

[25] D. Schmidt and S. D. Huston. *C++ Network Programming: Mastering Complexity with ACE and Patterns*, volume 1. Addison-Wesley, 2002.

[26] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.

[27] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of Java program. In *Proc. of the 8th International Workshop on Principles of Software Evolution*, 2005.

[28] M. W. van Someren, Y. F. Barnard, and J. A. C. Sandberg. *The Think Aloud Method*. Academic Press, London, 2004.

[29] A. M. Vans, A. von Mayhauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *Int. J. Hum.-Comput. Stud.*, 51(1):31–70, 1999.

[30] C. Wallace, C. Cook, J. Summet, and M. Burnett. Assertions in end-user software engineering: a think-aloud study. In *Proc. of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 63–65, 2002.