

An Initial Study of Common Coding Pitfalls in Java Programs*

Fancong Zeng

Department of Computer Science
Rutgers University
Piscataway, NJ, 08854
fzeng@cs.rutgers.edu

Abstract

This paper reports an initial study on classifying common Java coding pitfalls with an emphasis on the patterns that lead to such coding pitfalls. Documentation of such pitfall patterns helps to avoid making common mistakes and to shorten program debugging time.

1. Introduction

A lot of Internet applications are written in Java, a more and more popular programming language for the Internet. Program testing and debugging are difficult; using multiple threads in most of these Internet applications introduces new difficulties to their testing and debugging. One difficulty is nondeterminism: different runs of the same program with the same input may have different output. Most common bug types of multithreaded Java programs, e.g., deadlocks, livelocks, and race conditions, are related to nondeterminism.

Tools have been developed to detect bugs in Java programs, e.g., [1] and [2]. However, these tools are lack of an important property: localization of bugs. Tools based on static analyses can produce many false positives when processing large programs. In addition, in the case of model checking, mapping back a counterexample to the source code is a difficult task. Tools based on test cases can report bugs but they usually do not report where in the programs the bugs are.

It is difficult to find a needle in a sea, but it is easy to see a camel in a classroom. Similarly, in order to effectively localize bugs, a necessary technique is to restrict bugs to a limited scope. This paper proposes a novel approach for identifying as small as possible (Java) code regions that

likely contain bugs. The key idea of the approach is to identify and document patterns that lead to common coding pitfalls.

Outline The rest of the paper is organized as follows: section 2 describes the idea of using patterns for delimiting code regions that possibly contain bugs. Section 3 reports an initial study of common coding pitfalls in Java programs. Section 4 proposes a format for documenting pitfall patterns. Section 5 compares related work. Section 6 concludes current work and makes a perspective of future work.

2. Coding pitfalls and their patterns

A pattern is “a consistent, characteristic form, style, or method”[5]. I have the following observations with respect to patterns:

- When people perform a task, they usually follow some patterns, which are derived from their experience from practice and/or textbooks.
- Some people tend to follow the same patterns.
- Some patterns are good in the sense that they will lead to a success, while some patterns are bad in the sense they will lead to a failure.
- Usually a pattern instance exists within a small amount of time and space.
- Pattern instances are recognizable.
- Multiple pattern instances coexist well and they usually don't interfere with each other.

A coding pitfall is some code that is not considered good. A coding pitfall is likely a fault. A pitfall pattern is a pattern that makes a programmer produce coding pitfalls.

The idea of using coding pitfalls and pitfall patterns to delimit possible bug regions is as follows: suppose we have recognized coding pitfalls and their patterns. Since a coding pitfall is a potential fault, the union of these coding pitfalls is a scope of faults. This scope may still be large, but

*Proceedings of MASPLAS'03: Mid-Atlantic Student Workshop on Programming Languages and Systems, Haverford College, April 26th, 2003

it can be narrowed down by examining the coding pitfalls. For example, if a small region in that scope contains several coding pitfalls, this region is a good start place for debugging. Another good start place for debugging is a small region containing many coding pitfalls that are likely to be made by the programmer who wrote the program under examination. After the scope is narrowed down, tools can be exploited to test and debug the scope.

To make this novel idea effective in practice, a lot of issues need to be addressed. For example, What are the common coding pitfalls? How often does a particular coding pitfall occur? What should be included in a pitfall pattern? What is the pitfall pattern that causes a coding pitfall? How to represent a pitfall pattern? etc. In the next sections, I discuss these interesting issues.

3. An initial study of coding pitfalls

As an initial study, I chose some multithreaded programs as data points and focused on coding pitfalls related to multithreading. Below I introduce the background of the programs and programmers in the study, and report and analyze the common coding pitfalls found through the study.

3.1. Programmers and programs

The programs were an output of a class project of an undergraduate OS course at Rutgers in Spring 2002. In Spring 2002, the programmers were senior or junior students from Computer Science department or Computer Engineering department at Rutgers. They were not very familiar with Java, e.g., some of them did not know whether a Java program could deadlock or not. Thus, I classify them as Student Programmers.

The programmers were asked to solve two synchronization problems. One was the Many Readers/Single Writer Problem, the other was the “cigarette smokers problem”(pages 237-238, [8]). There were 36 usable programs written by 19 students: 18 for the cigarette smokers problem and 18 for the Many Readers/Single Writer problem.

3.2. Coding pitfalls w.r.t. multithreading

I did not use any bug-detection tools in finding pitfalls in the 36 programs. There are two reasons. One is that the tools I have collected do not have good usability, and the other is that the tools can find bugs but they cannot identify coding pitfalls. Thus, I tried to identify the coding pitfalls by inspecting the code. I found quite a few coding pitfalls w.r.t. multithreading, which are reported below with the number of occurrences:

1. *Exclusive Read* In the Many Readers/Single Writer problem, at one time at most one reader can read. (frequency: 4)
2. *No synchronization* No synchronization where synchronization is necessary. (frequency: 4)
3. *Use yield() instead of wait()/notify()* Using yield() instead of using wait() or notify() on condition variables.(frequency: 3)
4. *Wait without a condition* The wait() operation is not associated with a condition. (frequency: 3)
5. *Unnecessary locks* To use an unnecessary nested lock for protection. (frequency: 2)
6. *Wait() inside an if* Wait() is put inside an 'if' statement instead of a 'while' loop. (frequency: 1)
7. *Use sleep() instead of wait()/notify()* Using sleep() instead of using wait() or notify() on condition variables.(frequency: 1)
8. *Notify() without wait()* There is a notify() operation but there is no wait(). (frequency: 1)
9. *Notify() without a lock* Notify() is performed without obtaining the corresponding lock. (frequency: 1)
10. *Create a thread but does not start it* The wait() operation is not associated with a condition. (frequency: 1)

If one programmer made the same coding pitfall several times, I only count 1 in the frequency calculation. In addition to the frequencies, I record the relation between the coding pitfalls. Specifically, different pitfalls made by the same programmer are recorded.

I use tuples to denote the coding pitfalls made by the same programmer. 15 programmers made coding pitfalls, and the pitfalls they made in terms of tuples are: (1,3), (1,3), (1,3), (1,10), (2), (2), (2,4,5,6), (2,8), (4), (4,9), (5), and (7). One programmer that made a (2) did not have a useful program for the cigarette smokers problem, and the programmer that made a (7) did not have a useful program for the many readers/single writer problem.

The existence of one (1,10) shows that C programmers may take some time to adapt themselves to Java; several instances of (1,3), (7), (4,9), and (2,8) suggest that the programmers were not familiar with Java monitors, in particular, condition variables; most of the pitfall tuples suggest that it is not easy for student programmers to grasp multithreaded programming.

3.3. The E3 features

A further analysis shows that the approach of using coding pitfalls has the following E3 features:

- *Efficient* The coding pitfalls were identified by code inspection without running the program, and most of them were identified by syntactic checks plus lightweight semantic checks, for example, whether some method names are used. Furthermore, most these checks were performed in an intro-procedural fashion. Tools can be developed to help finding coding pitfalls.
- *Effective* All “Exclusive Read” pitfalls change the requirement of the problem; one “wait without a condition” and the two “unnecessary locks” are not faults; the rest pitfalls are faults. These faults may lead to deadlocks and race conditions. Moreover, programs are written by programmers. If a program fragment contains quite a few coding pitfalls the programmer tends to make according to history, this program fragment likely contains bugs.
- *Educational* Before they started to do the projects, the students were provided examples on how to create, start and join threads, and on how to invoke a wait()/notify() for a thread. Thus, the student did not make other common pitfalls like to join a thread too early, and the frequency of some pitfalls, e.g. “ wait inside an if”, is small.

The above E3 features suggest that coding pitfalls can be of great use for the purpose of testing and debugging. The next section proposes a template to document the pitfalls with an emphasis on their patterns.

4. Using patterns to document pitfalls

4.1. A documentation template

Influenced by [7] and [3], I use the following template for documenting coding pitfalls:

1. *Description* What is this pitfall?
2. *Name* What is this pitfall pattern called?
3. *Category* What category does the pitfall falls into?
4. *Language* Which programming language is used?
5. *Programmer Classification* What programmers are likely to make this pitfall?
6. *Problem* What problem does the programmers try to address when they make the pitfall?
7. *Context* What is the context of the problem?
8. *Coding Pitfall* What is parameterized pseudo-code for this pitfall?
9. *Possible Consequences* What are the possible consequences of this pitfall?
10. *Related pitfall patterns* What are the related patterns?
11. *Frequency* How often does the pitfall occur?

12. *Rationale* Why do programmers make this pitfall?
13. *Prescription* How to avoid making this pitfall?

The template is easy to be understood by programmers, as a few student programmers told me after using it. It contains useful information for using pitfall patterns in programming practice, as most of the fields are directly related to programming practice. It is not restricted to one language, as the programming language used is a field of the template.

After enough coding pitfalls are identified and represented, and the corresponding pitfall patterns are extracted, a pitfall pattern library can be built for a group of programmers. Such a library can be used to narrow down the fault scope of programs written by that group. It can also be used to teach programmers in that group how to avoid writing buggy programs.

The next subsection describes two pitfall patterns using this template.

4.2. Two pitfall patterns

Pitfall Pattern 1:

1. *Description* A potentially faulty wait because it is within an “if” instead of a “while”
2. *Name* Put_wait_inside_if
3. *Category* Multithreading
4. *Language* Java
5. *Programmer Classification* Student Programmers
6. *Problem* How to make a thread wait for some condition which can be changed by another thread?
7. *Context* There are several communicating threads in a system. In particular, one thread blocked on a condition variable can make progress only under a special condition, which can be made true or false by another thread.
8. *Coding Pitfall*
 Def x = a “wait()” occurrence in
 NOT(while (COND_FOR_MONITOR) { ...x...})
 AND (if (COND_FOR_MONITOR) { ...x...})
 where COND_FOR_MONITOR stands for a condition associated to a condition variable.
9. *Consequences* Programs may behave differently from what programmers want them to.
10. *Related pitfall patterns*
 Wait_without_Condition
 Lack_synchronization
 Abuse_of_yield_or_sleep
11. *Frequency* Small after an example is shown
12. *Rationale* not familiar with the flavor of “signal_and_continue”; not familiar with multithreading.
13. *Prescription* Illustrate concepts of monitors and multithreading; give examples because examples in this case are effective

Pitfall Pattern 2:

1. *Description* A potentially faulty wait due to lack of a condition associated to the condition variable
2. *Name* Wait_without_Condition
3. *Category* Multithreading
4. *Language* Java
5. *Programmer Classification* Student Programmers
6. *Problem* How to make a thread wait on a condition variable?
7. *Context* There are several communicating threads in a system. In particular, one thread blocked on a condition variable can continue execution only after it is notified by another thread.
8. *Coding Pitfall*
Def x = a “wait()” occurrence in
NOT(while (COND_FOR_MONITOR) {...x...})
AND NOT(if (COND_FOR_MONITOR) { ...x...})
AND (...x...)
where COND_FOR_MONITOR stands for a condition associated to a condition variable.
9. *Consequences* Deadlock
10. *Related pitfall patterns*
Put_wait_inside_if
Lack_synchronization
Abuse_of_yield_or_sleep
11. *Frequency* Medium
12. *Rationale* Assume an order of thread execution; mix the concept of semaphores with that of monitors.
13. *Prescription* Illustrate concepts of nondeterminism and monitors.

5. Related work

[4] reported a study of pitfalls on multithreaded C programs. Their study suggests that it is difficult for students to grasp multithreading. My study focuses on how to avoid common pitfalls and to shorten debugging time.

Some tools have been developed for debugging Java programs. Some of them do not run the program under examination, e.g., [1], while some of them run the program with test cases (and may exploit a customized JVM), e.g., citeB99. With the help of coding pitfalls and their patterns, the quality of these tools can be improved.

Design patterns have been studied for about 10 years[7], and there is also research on pattern-oriented software architectures[3]. The template to document pitfall patterns is influenced by design patterns and architectural patterns.

In a recent talk, Erich Gamma, a coauthor of [7], indicated that the main contribution of design patterns in the past ten years is to encourage and facilitate people to discuss and use good software designs [6]. I hope my work will arise people’s interests in communicating and avoiding imperfect and possible fault code.

6. Conclusion and future work

I performed and reported an initial study on common coding pitfalls. The idea is novel and promising, and there are many interesting open questions, e.g., what kinds of faults can be covered by coding pitfalls? How to give scores to faulty code scopes based on coding pitfalls? I expect to further explore the pattern-oriented fault detection approach to make it widely accepted in programming practice.

References

- [1] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In: D. Grant (Ed.), *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 2001)*, pp. 68 - 75, Canberra, Australia, August 2001. IEEE Computer Society, PR 01254.
- [2] Derek Bruening. Systematic Testing of Multithreaded Java Programs. *Master of Engineering Thesis*, Massachusetts Institute of Technology, 1999.
- [3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, published by published by John Wiley & Sons, Inc. ISBN: 0471958697, August 1996.
- [4] Sung-Eun Choi and E Christopher Lewis. A Study of Common Pitfalls in Simple Multi-Threaded Programs. In *Proceedings of the Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, March 2000.
- [5] Editors of The American Heritage Dictionaries, *The American Heritage Dictionary of the English Language*, 4th edition, published by Houghton Mifflin Co. ISBN: 0395825172, January, 2000.
- [6] Erich Gamma, Design Patterns: — Ten Years Later, In Software Pioneers, M. Broy, E.Denert (Eds), published by Springer-Verlag Berlin Heidelberg. ISBN: 3540430814, February 2002.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, published by Addison Wesley. ISBN: 0201633612, October 1994.
- [8] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating System Concepts*, Sixth Edition, published by John Wiley & Sons, Inc. ISBN 0471417432, June 2001.