Antipattern-based Detection of Deficiencies in Java Multithreaded Software

H. H. Hallal[1], E. Alikacem[1], W. P. Tunney[2], S. Boroday[1], A. Petrenko[1]
[1] *CRIM, 550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada*
[2] *SAP Labs Canada, 111 Duke St., Suite 2100, Montreal, H3C 2M1, Canada*
{Hallal, Alikacem, Boroday, Petrenko}@crim.ca
Patrick.Tunney@sap.com

Abstract

In this paper, we investigate an antipattern-based approach to analyze Java multithreaded (MT) programs. We present a library of 38 antipatterns, which describe predefined recognized sources of multithreading related errors in the code. The antipatterns are archived in practical, easy to use templates, and are classified according to their potential effects on the program behavior. We also report on our experience in using these antipatterns in the analysis of real multithreaded applications.

1. Introduction

Multithreading enables concurrent execution of several threads within the same program. Thus, it is a convenient way to decompose large programs into relatively independent smaller tasks and increase the overall efficiency [21]. This programming technique has been closely associated with Java, which was designed with multithreading in mind. There, the benefits of multithreading are obvious, especially in the case of multiprocessor systems. Actually, even when a system has a single processor, reasonable Java virtual machine systems can implement a scheduling scheme that shares the processor among the running threads. These benefits have driven large software development companies like SAP, where the developed applications are often web based and involve, e.g., extensive handling of server side resources, to adopt Java MT programming in the development phase.

However, despite all the benefits of multithreading, the development, testing, and maintenance of MT applications are often difficult tasks and consume a lot of efforts. This is due mainly to the fact that multithreading is associated with both concurrency and nondeterminism, which add to the complexity of the tasks in each of the phases of the development cycle. In fact, unlike sequential applications, a MT program can have several execution paths for the same input data because of the unpredictable thread scheduling and the inherent parallelism. Therefore, when an error occurs during one execution of a program, it might not be possible to reproduce the same error by simply re-executing the program (as in the case of sequential programs). At the same time, following a specific thread scheduling, an error may occur in only few of the possible executions of the program, thus making it difficult to detect. This makes traditional testing and debugging methods inefficient for the detection of multithreading related problems.

As a result, developing appropriate technologies, supported by tools, for identifying concurrency associated defects has become the focus of many experts and one of the most important problems in software engineering [14]. The hope is that through automation, the efforts spent on detecting errors can be reduced substantially. That is why many researchers around the world, currently concentrate on developing techniques and tools for testing and detecting errors in multithreaded applications. Many sophisticated formalisms, models and techniques have been introduced to serve these purposes. Unfortunately, such means are not always easily applicable and ready to be integrated into development processes, especially in industrial production of software. The reasons include the complexity of the techniques, the steep learning curve (to master modeling and reasoning methodologies) and the scalability in the case of large systems. Recently, some surveys, e.g., [14], appeared that aim at encouraging a worldwide cooperation to eventually develop MT application test tools. This indicates that one can anticipate an extended period of intensive research and development in the area of MT Java applications.

Meanwhile, numerous mistakes can either be detected or avoided by simpler means, especially when they are caused by erroneous syntactic constructions or known sloppy programming styles. Actually, there is high tendency in recent works to profit from the improvement that the concept of design patterns [12] has brought to the development of software applications. The main idea is to consider representing common errors (that are known to occur frequently in large number of applications or that are shared by several programmers) by patterns of errors that could be generalized in a way to facilitate their detection and refactoring. In this regard, the notion of an *antipattern*, "something that looks like a good idea, but which backfires badly when applied" [27], is being increasingly used in the testing and debugging phases. Current efforts include building libraries of antipatterns and developing tools that can detect them in software applications, in general, using both static and dynamic analysis approaches.

In this paper, we present a library of antipatterns that describe multithreading related problems to facilitate their detection in MT Java applications. We catalog as many as 38 antipatterns that relate to concurrency, synchronization, and some common MT programming techniques, which makes these antipatterns famous among programmers and testers. We present a practical classification of the antipatterns, which is less abstract than the widely known classifications; and we define an antipattern template (to archive the antipatterns), which provides useful information for the debugger and the developer as well. We use some existing tools, which detect antipatterns, and perform experiments to evaluate the relevance of the antipatterns as well as the adopted detection techniques. This work is a part of a joint collaboration between CRIM and SAP Labs Canada, which aims at finding appropriate techniques and tools to analyze MT Java programs being written in the Eclipse development environment.

The remainder of this paper is organized as follows. Section 2 presents an overview of the common problems encountered in MT applications and the existing solutions. In Section 3, we describe the antipattern-based approach to detect multithreading related defects. Then, in Section 4, we present the library of multithreading related antipatterns with the corresponding experimental results. Then, conclude the paper, draw the main conclusions, and discuss the potential future enhancements in Section 5.

2. MT Problems and Detection

It is known that proper communication and coordination between threads are not simple to realize, especially when an application grows large in size and scope (i.e., it has many threads, which could be active at the same time). Therefore, it is not always easy to write MT programs that are free of problems, which relate directly or indirectly to both the correctness and performance. In the following, we review the most common problems that could result from the misuse of concurrency in MT applications.

Races. Broadly stated, races occur when several threads access the same resource simultaneously without proper coordination [6, 24]. As a result, the program might end up producing an output far different from the one desired.

Deadlocks. There are several definitions of a deadlock in MT applications. However, they all share one fact: when a deadlock occurs in a program, the whole program or part of it is not alive anymore. According to [5], "a deadlock is an impasse that occurs when multiple threads are waiting for the availability of a resource that will not become available because it is being held by another thread that is in a similar wait state".

Livelocks. A livelock occurs when one thread takes control (e.g., locks an object of a shared resource) and enters an endless cycle. This is usually hard to detect since it is not easy to distinguish between a long cycle and an endless cycle. More generally, it occurs when a thread is so busy accepting new work that it never has a chance to finish any tasks [19]. The thread could eventually exhaust the memory and cause the program to crash. In [5], "a livelock is a condition in which two or more threads continuously change their state in response to changes in the other thread(s) without doing any useful work".

Efficiency Problems. As much as it is needed in MT programs, synchronization causes a significant overhead that usually accounts for 5-10% of the total execution time [1]. This follows from the fact that managing synchronization in Java MT applications requires the Java Virtual Machine (JVM) to perform some internal tasks (e.g., writing any modified memory locations back to main memory) that could impair the efficiency of the application.

There is a large body of research involved in trying to improve the quality of multithreaded programs both in academia and in industry. Progress has been made in many domains and it seems that a solution will contain components from many of them. As [14] states, there is no silver bullet solution; so research addresses a variety of partial solutions. The existing detection algorithms could be broadly divided into static and dynamic.

Static algorithms. They are based on the analysis of code (source code or bytecode) and are normally independent of input order or thread scheduling since the code is analyzed without execution. Static analysis tools of various types, including formal analysis tools, are being developed, which can detect faults in the multi-threaded domain [1, 13, 14, 26, 29]. Analysis tools that show a view of specific interest in the interleaving space both for coverage and performance [28] are also being considered. Meanwhile, additional input data, such as annotations, could also be used.

Dynamic algorithms. They require the execution of the program and depend on the analysis of the resulting execution traces. While dynamic analysis usually generates fewer false warnings than static algorithms do, it often restricts error detection to faults that manifest themselves in or could be only deduced from observed traces [10, 20, 22, 24].

3. Antipattern-based Approach

The concept of patterns has become widely adopted in the development process, especially in software applications. Design patterns, which are defined as insightful and clever ways of solving particular classes of problems, were introduced in the last decade to save on the cost of producing and maintaining software products [12]. Recently, in the context of program validation and error detection, the concept of predefined error description has been introduced to help reduce the efforts associated with debugging and maintaining of software applications [25]. Two types of antipatterns are identified so far.

Design antipatterns. Common software designs that have been proven to fail repeatedly, i.e., models of syntactic constructions (in particular) representing potential or confirmed sources of problems in a program [25]. From this viewpoint, an antipattern is simply a solution to a problem that does not work correctly, and it can be seen as just another design pattern [4].

Error or bug patterns. A bug pattern represents a recurring correlation between signaled errors and underlying bugs in a program [2]. Unlike design antipatterns, bug patterns are patterns of erroneous program behavior correlated with programming mistakes. The concern is not with design, but with the programming and debugging process from the viewpoint of symptoms of errors [2]. Meanwhile, quality problems and inefficiencies are not covered by bug patterns [2].

Note that there exist some interpretations that do not distinguish between the two notions, e.g., [15] uses the notion of error or bug pattern to identify problems from both categories. In our case, the analysis of MT programs is based merely on the source code. Therefore, the notion of bug patterns as mentioned by [2] might not be directly applicable since the symptoms of errors (depicted in the behavior of the program) cannot be easily compared due to concurrency and uncontrolled scheduling. Actually, as we mentioned earlier, reproducing the same error, when it occurs in an execution of a program, is not an easy task. That is why, based on the symptoms associated with an error (or an error antipattern) it is difficult to reason about the causes of that error. Consequently, in our work, we consider antipatterns as members of the first category, i.e., representing models of syntactic constructions and design options, which constitute potential sources of erroneous behavior.

In the literature, there have been several attempts to use predefined error formulation to detect the problems that relate mostly to multithreading and concurrency [9, 11, 15, 17, 18]. However, we are not aware of any attempts to build a library that archives all these problems and represents them in an easy to use way (from the viewpoint of the developer who deals mainly with the code). In the following, we discuss our work in constructing a library of the most common errors and problems encountered in MT programs. First though, we overview the basic detection techniques that are commonly used to implement an antipattern-based approach in general and the main hurdles that are faced.

The errors that can be coded in antipatterns are usually established while considering the commonly met mistakes, the expertise in the domain of the development of MT applications, and the constraints related to the language semantics. Often, also, errors are caused by inadvertency or lack of experience, mainly in the case of novice programmers. Therefore, detection is commonly done using static analysis since the majority of antipatterns can be represented by syntactic constructions. Here, we can distinguish between two classes of techniques:

- Detection by linear scanning of the source code: This requires a browsing of the syntactic tree of a program without considering its control flow.
- Detection using some abstractions extracted from the source code: This requires supplementary analysis of extracted information, which could include call graphs, class dependency graphs, etc.

However, static analysis is known to yield a non-negligible number of false alarms, mainly because some necessary dynamic information is usually estimated (often in a pessimistic way in order to get safe results). In addition, an antipattern itself can be a source of false alarms since, in some cases, it models a construction that could potentially, but not necessarily, produce an error. Meaning that even when the detection is correct (the antipattern is found), the doubtful construction can still prove to be problem free, especially when it concerns problems that are related to style and efficiency. Moreover, when a detection technique depends on abstractions or information extracted from the code, imprecision is almost inevitable, thus becoming an additional source of false alarms. For such reasons, dynamic analysis can be the only or the last resort for the detection of certain antipatterns, which are manifested only when the program is executed and cannot be detected unless some dynamic information is available.

In conclusion, the outcome of applying an antipattern-based analysis approach depends on several factors:

- The relevance of the error represented by the antipattern. It is necessary that the antipattern represent an error that is relevant to the application under test.
- The correct formulation of the antipattern, meaning that it models correctly the error.
- The soundness of the detector to ensure it does not generate many false alarms.

4. Multithreading Antipattern Library

Some would argue that an antipattern library could never be complete. Actually, as long as antipatterns are related to programming styles (which are not well defined themselves), one may keep coming up with new additions to a library. Here, we present a library of the problems most commonly considered in the literature and by authors of several analysis tools. We reviewed existing classifications in the research literature, the practical programming guidebooks, and the existing tools that target the program analysis problem. Throughout our search for antipatterns, we had in mind to catalog problems, errors, and programming styles that are both interesting to and understandable by the developers of the target applications. So far, this library includes 38 multithreading related antipatterns and is in continuous evolution, because some antipatterns can become obsolete or controversial while new ones can appear.

4.1 Classification

So far, problems that could be encountered in distributed applications, in general, are categorized and grouped mainly by considering the abstract effects of concurrency on the developed programs. Hence, experts distinguish mainly between *safety* and *liveness* problems that could be faced in a program. This categorization is generally based on the abstract definitions of safety (nothing bad will ever happen) and liveness (something good will eventually happen). Safety problems usually affect the integrity of the handled data in a program, e.g., race conditions. On the other hand, liveness covers problems leading to deadlocks and livelocks. Despite its theoretical soundness, this categorization becomes too abstract and vague for developers and programmers when analyzing MT applications on the level of the written code. Meanwhile, other classifications have been proposed like the one adopted in the Specification Pattern System (SPS) [7], which distinguishes between order and occurrence patterns. However, this classification, as the SPS itself, originates from an understanding of the analyzed application that takes into account the functional aspects of the system under test and not the correctness and quality of the written code. We mention also the classification that distinguishes between static and dynamic problems following the techniques used in the detection of the problems. Another possibility, which we find closer to the understanding of the developer, is to classify the problems of a multithreaded application in the following two main groups:

Correctness problems. They affect the outcome of the program. Either the program produces a wrong output (data and memory corruption) or it does not produce any output at all due to deadlocks, livelocks, race conditions, or memory exhaustion.

Efficiency and quality problems. They affect mainly the performance of the program, so the desired output could still be produced but not in the desired time. Also, the problem could be that the program is not easily usable or maintainable.

On a lower level, and to further simplify the task of the developer who needs to detect and solve the problems, we identify the following classes of MT related antipatterns:

1. Deadlocks
2. Livelocks
3. Race Conditions
4. Efficiency Problems
5. Quality and Style Problems.
6. Problems Leading to Unpredictable Consequences.

Note that the last category groups the antipatterns that are known to occur in MT applications, but whose outcome is not directly identified in the behavior of the application. For example, the `start()` method of a thread should not be called more than once. However, if this occurs in a program, there is no indication of how the application will react (sometimes this will generate an exception at runtime, but in other cases the second call will be ignored [21]). Another antipattern that could cause similar confusion is the inconsistent synchronization antipattern [15]. This antipattern is used to identify potential data races by detecting attributes that are defined and used with and without synchronization in the tested application. However, this antipattern can be used to detect quality and efficiency related defects in the application as well. Actually, when an attribute that does not need synchronization is used with and without it, the potential problem is no more related to races, but one that affects the efficiency or the quality of the program. For this reason, we archive this antipattern with the problems having unpredictable results since there are no clear indications, in the definition of the antipattern and the proposed detection methods, how to interpret its presence in a program.

At the same time, there are antipatterns, which result in problems in more than one aspect of the program behavior. For example, the unsynchronized spin-wait (mainly an unsynchronized loop) may result simultaneously in a livelock -the loop might never exit- and in deteriorating the efficiency of the program. In our library though, we archive the antipatterns in the category that is known to have the most serious consequences on the analyzed program. Therefore, in the case of the unsynchronized spin-wait antipattern, we archive it in the livelock category.

4.2 Template

To archive the antipatterns in our library, we define the following template. This template provides useful information about a particular antipattern including:

- the definition (name, description, and category),
- an example of occurrence (code samples),
- the re-factoring solution (hints and tips),
- possible conflicts of applying the proposed solutions,
- the possible detection technique(s), and
- comments (e.g., source of the antipattern).

This information, in the template, is meant to help both the developers of MT applications and the professionals building tools to detect the antipatterns in MT programs. For the latter, information like possible detection techniques contributes to the reduction of the implementation efforts, especially the examples how an antipattern would be manifested in the source code. As for the former, the examples, the potential conflicts of any proposed solution, as well as the comments could optimize the application of the proposed refactoring solution or even serve as means to avoid the antipattern in the first place.

Note that our classification and documentation of antipatterns can be compared to the work in [8], where a classification of concurrent bugs is suggested based on the causes of a bug, e.g., wrongful assumption. Three bug classes corresponding to wrong assumptions on protection, interleavings, and liveness are suggested. While several preliminary bug examples are described, the work does not target systematic documentation of these bugs. However, in our case, both the classification and the documentation are meant to facilitate the task of a developer in debugging and correcting his code.

4.3 Antipatterns

Here, we present the antipatterns that we have cataloged so far in our library classified in their corresponding groups. We also report, for each class of antipatterns, our experience in using the antipatterns. Actually, we have used some of the antipatterns in the validation of MT Java code of several applications in the context of the joint collaboration between CRIM and SAP Labs Canada. For this purpose, we have used two existing tools, which analyze byte code of MT programs: Jlint [3, 17] and FindBugs [9, 15]. The two tools target the detection of many antipatterns (12 and 14, respectively) from the proposed library. In addition, we have developed a third tool, E_Jlin, which analyzes the source code using linear scanning techniques and detects nine antipatterns so far. This tool is based on the existing analysis tool, Jlin, developed by SAP as a plugin in the Eclipse development environment.

Due to space limitations, we only show the names of the antipatterns (with explanations when mostly needed), and next to each antipattern, we list the tools that can detect it where available. Then, we give a brief example from each class of antipatterns. In our experiments, we considered three applications that use multithreading heavily and whose developers are available to discuss and evaluate the results. The first application is a Multi-Agent Platform developed at CRIM; the second is a SAP Vending Software, and the third is a UDDI server also developed by SAP.

Deadlock

1. Synchronized method call in cycle of lock graph, (Jlint).
2. Method call sequence leads to a cycle in lock graph, (Jlint).
3. Cross Synchronization, (Jlint).
4. Method `wait()` invoked with another object locked, (Jlint).
5. Call sequence can cause deadlock in `wait`(), (Jlint).
6. *`Identifier`*.`wait()` method called without synchronizing on *`identifier`*, (Jlint).
7. Unconditional `wait()`: not testing the condition for which a wait is needed, (E_Jlin, FindBugs) .
8. Unconditional `notify()` or `notifyAll()`, (E_Jlin, FindBugs).
9. Waiting forever: a thread executes `wait()` but is never notified.

Among the tools used, only Jlint reported error messages. Actually, Jlint detected two antipatterns in App1, but it crashed with the other two applications:

- Synchronized method call in cycle of lock graph.
- Method call leads to a cycle in lock graph.

Note, though, that for the detected antipatterns, the number of messages reported was 367 and 152, respectively (which is quite high). The analysis of the results showed that the high number was due mainly to the conservative approach followed by Jlint to determine the Lock Graph, on which the detection of deadlock antipatterns is based. The other reason was the way Jlint represents results. Actually, when a method is found to cause an error, the tool generates a message for every call to the method as well.

Name: Synchronized method call in cycle of lock graph

Description: The lock dependency graph is a graph whose nodes are classes and arcs lock acquisitions between the classes. When a synchronized method appears in a cycle of the lock graph, it could indicate that calling this method could lead to a deadlock in the application.

Category: Deadlock

Example:

```
public class A
{ Object b = new B();
  public synchronized void foo()
   { b.bar(); } }
public class B
{ Object b = new A();
  public synchronized void bar()
  { a.foo(); } }
class MyThread implements Runnable {
  public void run()
  {...  a.foo(); ... }
}
```

Detection: Compute the lock graph and detect the cycles in the lock graph. Then, identify the synchronized methods that make part of the cycles. This antipattern is detectable by Jlint.

Refactoring: Proper reordering of lock acquisition among the threads involved in the deadlock.
Conflicts: This solution might lead to data races.

Comments: Detection highly depends on the expressiveness of the computed lock graph.
Source: http://artho.com/jlint/manual.shtml

Figure 1. Synchronized method call in cycle of lock graph.

Livelock

1. Unsynchronized spin-wait, (E_Jlin, FindBugs).
2. Wait stall: a thread calls `wait()` without specifying a timeout, (Jprobe Threadalyzer).

None of the livelock antipatterns was reported by any of the tools. This could reflect either the freedom of the applications from these antipatterns or the complexity of detecting them. The latter could be the case, in particular, to the "Unsynchronized spin-wait" antipattern. The tools do not report its presence in the applications, especially in App1, which was developed mainly by students. However, the antipattern is known to occur frequently in MT applications, especially developed by beginners, and it is important to detect it because it severely decreases the performance and affects the correctness of an application.

Name: Unsynchronized spin-wait   AKA: Spin-wait

Description: It appears in the form of an unsynchronized loop, whose exit condition is controlled by another thread. Resulting problems include exhaustive use of resources (CPU time) and thread stalls.

Category: Livelock

Example:
```
//wait for spider to finish
while (spider.value != 10)
{ //do nothing, just test again   }
```

Detection: For the basic instances of the pattern, as illustrated in the example: Detect in the code all occurrences of empty loops with attributes as condition variables.
This antipattern is detectable by FindBugs.

Refactoring: (For simple cases as in the example) Use `yield` or `sleep` to take control from the current thread. This will provide other threads (especially lower priority ones) with a chance to execute while the condition is not yet fulfilled.
Conflicts: Improper use of synchronization might lead to deadlocks or data races.

Comments: This antipattern has several variations.
Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", http://www.cs.umd.edu/~pugh/java/bugs/

Figure 2. Unsynchronized spin-wait.

Race Conditions

1. Overriding a synchronized method, (Jlint).
2. Non synchronized method called by more than one thread, (Jlint).
3. Unprotected non volatile field used by several threads, (Jlint).
4. Non synchronized `run()` method, (Jlint).
5. The double-check locking for synchronized initialization, (E_Jlin, FindBugs).
6. Get-Set methods with opposite declarations, (FindBugs).

Here, only FindBugs and Jlint reported errors in the tested applications. FindBugs detected only one antipattern, the "Get-Set methods with different declarations", which was a confirmed error in App1. Jlint, on the other hand, detected four antipatterns (again only in App1):

- Overriding a synchronized method (one message).
- Non volatile field used by more than one thread (five messages).
- Non synchronized `run()` method (one message).
- Non synchronized method called by more than one thread (314 messages).

Name: Non synchronized `run()`

Description: When different threads are started for the same object that implements the Runnable interface, the method `run()` must be synchronized.

Category: Race condition.

Example:
```
Class MyThread implements Runnable{
   ...   public void run() {...} }
...
MyThread t1 = new MyThread() ;
New Thread(t1).start() ;
New Thread(t1).start() ;
```

Detection: Detect the creation of Java threads (Thread Class) with the same Runnable object, an instance of a class that implements the Runnable interface.
Data flow analysis could be used to determine which threads are created with the same Runnable instance.
This antipattern is detectable by Jlint.

Refactoring: Declare `run()` synchronized.
Conflicts: Oversynchronization.

Source: http://artho.com/jlint/manual.html

Figure 3. Non synchronized `run()`.

The messages reported by Jlint are interpreted as follows. Among the antipatterns with the low number of messages reported, only one false alarm was generated (in the case of the non synchronized run() method). As for the "Non synchronized method called by more than one thread" antipattern (314 messages), approximately only 12% of the generated messages were confirmed errors. The remaining messages are either redundant (Jlint produces a message each time a problematic method is called) or truly false alarms.

Efficiency Problems

1. Overuse of synchronized methods: a method is synchronized even if used by only one thread.
2. Synchronized read only methods while there are no methods with a write access.
3. Internal call of a method, (E_Jlin).
4. Object locked but not used: it is likely that synchronization is not needed (E_Jlin).
5. Excessive synchronization: a synchronized method contains several operations that do not need synchronization.
6. Overthreading: defining too many threads.
7. Irresponsive or slow interface (Complex computation in an AWI/Swing thread.)

8. Misuse of `notifyAll()`: `notify()` could be safely used, e.g., in the case of an object shared by only two threads.
9. Synchronized immutable object, (E_Jlin).
10. Synchronized atomic operations.
11. Calling `join()` to an immortal thread, (FLAVERS).

Name: Internal call of a method.

Description: Java allows reentrant monitors. One thread can get the monitor (lock) several times in a nested way. Aside from the first acquisition of the lock, synchronization is not necessary.

Category: Efficiency Problem.

Example:
```
Class Reetrant {
Synchronized void foo()
{this.bar();}
synchronized void bar()
{ ... }
... }
```

Detection: Use a call graph and data flow analysis.
This antipattern is detectable in E_Jlin.

Refactoring: Potential solutions
- Declare the called method (`bar()` in the example) non synchronized.
- Inline the called method, `bar()` in this example.

Conflicts: Risk of data races or corruption.

Source: J. Aldrich, C. Chambers, E. Gün-Sirer, S. J. Eggers. Static Analysis for Eliminating Unnecessary Synchronization from Java Programs.

Figure 4. Internal call of a method.

In the case of efficiency related problems, Jlint does not detect any antipatterns. All the antipatterns detected by E_Jlin (Table 1) and FindBugs (Misuse of `notifyAll()`: 1 message in App2) were confirmed problems in the three applications, and corresponded mainly to the overuse of synchronization, which affects the performance of an application.

Table 1. Efficiency problems detected by E_Jlin.

Application	E_Jlin
App1	• Internal call of a method (18) • Synchronized immutable object (1)
App2	• Object locked but not used (2) • Synchronized immutable object (2)
App3	• Object locked but not used (7) • Internal call of a method (7)

Quality and Style Problems
1. `wait()` is not in loop, (E_Jlin, FindBugs).
2. Reference value is changed when it is used in synchronization block, (Jlint).
3. Blob thread: a thread that implements most of program functionality.
4. Unnecessary notification: when no threads are waiting.
5. Premature `join()` call: joining a thread which has not started yet, (FLAVERS).
6. Dead interactions: calling already terminated threads, (FLAVERS).

Name: `wait()` is not in a loop

Description: While it is possible to correctly use wait() without a loop, such uses are rare and worth examining, particularly in code written by developers without substantial training and experience writing multithreaded code.

Category: Quality and style problem.

Example:
```
If (! Resource.available) {
     wait() ;  }
```
When the thread wakes up, it is better to check again the condition.

Detection: Check if the `wait()` method invocation is in a loop.
This antipattern is detectable by FindBugs.

Refactoring: Use while instead of if

Conflicts: None.

Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", http://www.cs.umd.edu/~pugh/java/bugs/

Figure 5. `wait()` is not in a loop.

In this category, only Jlint detected one problem in App1: Reference value is changed when it is used in synchronization block. The tool generated two messages for this antipattern and they both corresponded to confirmed errors.

Problems Leading to Unpredictable Results
1. Double call of the `start()` method of a thread, (E_Jlin).
2. `Start()` method call in constructor, (FindBugs).
3. Inconsistent synchronization: fields that are used both with and without synchronization in the application class, (FindBugs).
4. Improper method calls: e.g. deprecated methods, use of Thread unsafe methods, (FindBugs).

The antipatterns detected from this category were all confirmed occurrences of the respective antipatterns except for the "Double call of the `start()` method of a thread", which was a false alarm. This antipattern was detected by E_Jlin in App1. The problem with such antipatterns resides in the fact that tools tend to report an error when the `start()` method is called for the same thread identifier more than once. But, it could occur, as in the case of App1, that the identifier would receive a new thread object and the method would not be repeated.

Name: Double call of the `start()` of a thread

Description: The `start()` method call is not supposed to be used more than once for the same thread.

Category: Problem with unpredictable results.

Example:

```
 ...
t = new Thread(runnable) ;
t.start() ;
...
t.start() ;
```

Detection: For each creation of an instance of a thread (Thread Class), check if there are more than one call to `start()` method. Sometimes, the detection of the second invocation needs data flow analysis or a points-to analysis. This antipattern is detectable by E_Jlin.

Refactoring: Remove the additional unneeded `start()`.
Conflicts: None.

Source: Patrick Naughton, "The Java Handbook", McGrawHill, 1996.

Figure 6. Double call of the method `start()` of a thread.

The other errors were detected by FindBugs, and they were distributed among the three applications (Table 2). The `Start()` method call in constructor messages were confirmed errors while those referring to inconsistent synchronization (where it is not known if the problem is a data race or excessive synchronization) could not be tracked to their causes since FindBugs does not indicate the location of the error in the application.

Table 2. Antipatterns leading to unpredictable results detected by FindBugs.

Application	FindBugs
App1	Inconsistent synchronization (6)
App2	• Inconsistent synchronization (2) • `Start()` method call in constructor (2)
App3	Inconsistent synchronization (5)

5. Conclusions

In this paper, we presented a library of antipatterns for MT Java applications. This library is built from antipatterns that are known to occur frequently in the development of MT applications. We have cataloged 38 antipatterns that relate to multithreading, concurrency, and synchronization. Then we have classified the collected antipatterns following a categorization, which reflects the effect of the antipatterns on the outcome of the tested program. Under the two main categories of correctness and efficiency and quality, we classified the antipatterns into six groups: Deadlock, Livelock, Race Conditions, Efficiency, Quality and Style, and Antipatterns with Unpredictable Results.

Then, we constructed a library of the antipatterns, which were represented using a template that reveals useful information for both documentation and detection (correction) purposes. Finally, we conducted some experiments with known existing tools to evaluate the relevance of the antipatterns from the various categories to some industrial applications, which use multithreading heavily. In this context we have developed a new tool that uses linear scanning to analyze the code of the tested application. This tool has been integrated in the Eclipse environment and is subject to further development.

We believe that such a library can be helpful in many ways in improving the development and test of Java MT applications. We have implemented a prototype tool to detect antipatterns from different categories and integrated this tool in the Eclipse environment, adopted by SAP. This constitutes an efficient way to enhance the analysis of the MT applications as they are developed by programmers of different levels of expertise. On the other hand, as other research groups have proven, the antipatterns could be detected by independent, stand-alone tools, which may be applied to desired applications at different stages of the development process.

At a different level, antipatterns, especially those which relate to the controversial aspects of the (Java) programming language, could be used by experts who work on refining the language and the corresponding resources (JVM, JMM, etc.), to identify such controversies and address them in future releases of the language. So far however, the Java language has kept a high level of backward compatibility, i.e., all the antipatterns in our library are valid for all the available releases (even the latest beta version of J2SE [16]), and are expected to remain so for quite a long time.

This work could be enhanced in more than one direction. In addition to collecting and documenting more antipatterns, we plan on developing more tools to detect as many antipatterns as possible to facilitate the analysis of applications. The main concern here is to identify the best detection techniques to reduce false alarms. Then, the library itself could be integrated in the development environment (Eclipse in our case) as a help component to increase the usability of the antipatterns.

References

[1] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers, "Static Analyses for Eliminating Unnecessary Synchronization from Java Programs", *In Proc. of the 6th International Static Analysis Symposium*, Springer-Verlag, LNCS 1694, Venice, Italy, September 1999, pp. 19-38.

[2] Eric E. Allen, "Bug patterns: An introduction Developer Works", [online], February , 2001 http://www-106.ibm.com/developerworks

[3] Cyrille Artho, *Finding faults in multi-threaded programs*. Master's Thesis. Institute of Computer Systems, Federal Institute of Technology, Zurich-Austin. 2001.

[4] W. Brown, R. Malveau, H. McCormick, T. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, NY, 1998.

[5] M. C. Baur, *Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs,* Diploma Thesis, Computer Systems Institute, Swiss Federal Institute of Technology, Zurich, April, 2003.

[6] Jong-Deok Choi, Harini Srinivasan, "Deterministic Replay of Java Multithreaded Applications", *ACM SIGMETRICS Symposium on Parallel and Distributed TOOLS (SPDT),* ACM Press, August 1998.

[7] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. "Property Specification Patterns for Finite-state Verification". *In Proc. of the 2nd Workshop on Formal Methods in Software Practice*, Florida, March 1998.

[8] E. Farchi, Y. Nir, and S. Ur, "Concurrent Bug Patterns and How to Test Them", *In Proc. of the International Parallel and Distributed Processing Symposium* (*IPDPS'03*), Nice, France, 2003.

[9] FindBugs, http://www.cs.umd.edu/~pugh/java/bugs

[10] Cormac Flanagan, Stephen N. Freund, "Detecting Race Conditions in Large Programs", *In Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Utah, June 2001.

[11] FLAVERS, http://laser.cs.umass.edu/tools/flavers.html

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Boston, MA, 1995.

[13] P. Godefroid, "Model Checking for Programming Languages using VeriSoft". *In Proc. of 24th ACM Symposium on Principles of Programming Languages*, France, 1997, pp. 174–186.

[14] K. Havelund, C. Artho, D. Drusinsky, A. Goldberg, M. Lowry, C. Pasareanu, G. Rosu and W. Visser, "Experiments with Test Case Generation and Run-time Analysis", *In Proc. 10th International Workshop on Abstract State Machines (ASM'03)*, Taormina, Italy, March, 2003.

[15] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", [Online], http://www.cs.umd.edu/~pugh/java/bugs/

[16] J2SE 1.5.0 Beta 1, http://java.sun.com/j2se/1.5.0/index.jsp

[17] Jlint, http://artho.com/jlint/index.shtml

[18] Jprobe, http://www.quest.com/jprobe

[19] Neel V. Kumar, "Multi-threading in Java programs. Developer Works", [Online], March 2000. http://www-06.ibm.com/developerworks/java/library/j-multithreading.html

[20] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm, "Putting Static Analysis to Work for Verification: A Case Study". *In Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, Oregon, 2000, pp. 26-38.

[21] Patrick Naughton, *The Java Handbook*, McGraw-Hill, California, 1996.

[22] R. Netzer and B. Miller, "Detecting Data Races in Parallel Program Executions", *in Languages and Compilers for Parallel Computing*, ed. D. Gelemter, T. Gross, A. Nicolau, and D. Padua, MIT Press, 1991, pp. 109-129.

[23] Martin Rinard, "Analysis of Multithreaded Programs", *In Proc. of 8th International Static Analysis Symposium*, Paris, France, July 2001.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs" *ACM Transactions on Computer Systems*, ACM Press, NY, 1997, pp. 391-411.

[25] Connie U. Smith and Lloyd G. Williams, "Software performance antipatterns", *In Proc. of Workshop on Software and Performance*, Ottawa, 2000, pp.127-136.

[26] Scott D. Stoller, "Model-Checking Multi-Threaded Distributed Java Programs", *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, October 2002, pp. 71-91.

[27] Wikipedia, Antipatterns [Online] http://c2.com/cgi/wiki?AntiPattern

[28] A. S. Cheer-Sun Yang and L. Pollock, "All-du-path coverage for parallel programs". *In proc. of ACM SigSoft International Symposium on Software Testing and Analysis (ISSTA 98)*, Florida, March 1998, pp. 153–162.

[29] Yichen Xie and Dawson Engler, "Using Redundancies to Find Errors", *IEEE Transactions on Software Engineering*, 29(10), October 2003, pp. 915-928.