

1. ANTIPATTERN TEMPLATE

We propose the following template to represent antipatterns of problematic situations in the MT Java code. Each template provides information about a particular antipattern including the definition (name, description, and category), an example of occurrence (when possible), the re-factoring solution, potential conflicts of applying the solution, possible detection technique(s), and some comments.

<i>Name</i>	A concise definition of the problem.
<i>Description</i>	The situations in which this problem could appear. The effects it has on the code and the application.
<i>Category</i>	Deadlock, Livelock, Race Condition, Efficiency problem, Quality and Style Problem, Problem with unpredictable consequences.
<i>Example</i>	If available, sample code where the problem is illustrated.
<i>Detection</i>	How to detect the problem in the Java code. A high level description of the proposed algorithm to be used in the detection process.
<i>Re-Factoring</i>	Solution: How to solve the problem once detected in the program. Conflicts: Sometimes solving one problem of a certain class can cause another problem of a different class. For example, Blob threads and over synchronization.
<i>Comments</i>	The source of this pattern. Any comments that could be helpful in the detection or re-factoring.

2. ANTIPATTERNS

Some would argue that an antipattern library could never be completed. Actually, as long as antipatterns is 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 antipatterns that includes the antipatterns most commonly considered in the literature and by analysis tools. We have included as much details as possible in the templates so that they could be reused as documentation (help) in the tool environment.

2.1 Synchronized method call in cycle of lock graph

Name	Synchronized method call in cycle of lock graph
Description	The lock dependency graph is a graph whose nodes are classes and arcs are the lock acquisitions between the classes. When a synchronized method appears in a cycle of the lock dependency graph, it could indicate that calling this method could lead to a deadlock in the application.
Category	Deadlock
Example	<pre> 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 { Run() {... a.foo(); ... } </pre>
Detection	<ol style="list-style-type: none"> 1. Compute the lock graph 2. Detect cycles in the lock graph 3. Identify synchronized methods that make part of the cycles <p>Detectable by JLint.</p>
Re-Factoring	<p>Solution: Proper reordering of lock acquisition among the threads involved in the deadlock.</p> <p>Conflicts: Reordering the lock acquisition might lead to data races.</p>
Comments	<p>Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm</p> <p>The detection results highly depend on the expressiveness of the computed lock graph. Moreover, the presence of a cycle in the lock graph is a necessary condition for deadlock, but not a sufficient one. Therefore, there is high risk of numerous false positives.</p>

2.2 Method call leads to a cycle in lock graph

Name	Method call leads to a cycle in lock graph
Description	When a cycle in the lock dependency graph is reachable from a method call directly or through a chain in the call graph, the cycle could lead to a deadlock. This antipattern is the result of bad synchronization between threads of an application.
Category	Deadlock.
Example	<pre> 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 { run() {... method(); ... } ... method() { ... a.foo();...} </pre>
Detection	<ol style="list-style-type: none"> 1. Compute the lock graph 2. Compute the call graph For each cycle in the lock graph, detect method calls that lead to the cycle.
Re-Factoring	<p>Solution: Proper reordering of lock acquisition among threads involved in the deadlock.</p> <p>Conflicts: Reordering the lock acquisition might lead to data races.</p>
Comments	<p>Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm</p> <p>The detection results highly depend on the expressiveness of the computed lock graph. Moreover, the presence of a cycle in the lock graph is a necessary condition for deadlock, but not a sufficient one. Therefore, there is high risk of numerous false positives.</p> <p>Note: This antipattern can be seen as an extension of the "Synchronized method call in cycle of lock graph" antipattern.</p>

2.3 Cross synchronization

Name	Cross synchronization
Description	One thread nests its synchronization blocks in an order contrary to that of another thread vying for the same monitors, a deadlock can result. Each thread is stuck at the point indicated by the deadlock comment and is waiting for the release of the monitor that the other one already has.
Category	Deadlock.
Example	<pre> HashSet first = new HashSet(); HashSet second = new HashSet(); public void add(Object in) { synchronized (first) { synchronized (second) { first.add(in); second.add(out); } } } public void remove(Object out) { synchronized (second) { synchronized (first) { first.remove(in); second.remove(out); } } } </pre>
Detection	<ol style="list-style-type: none"> 1. Compute the lock dependency graph. 2. Detect the cycles in the lock dependency graph. 3. Identify the corresponding synchronized blocks. <p>Detectable by JLint.</p>
Re-Factoring	<p>Solution: Proper reordering of lock acquisition among the threads involved in the deadlock.</p> <p>Conflicts: Reordering the lock acquisition might lead to data races.</p>
Comments	<p>Source: Http://www.zdnet.com.au/builder/program/java/story/0,2000034779,20271828,00.htm</p> <p>It may involve more than two threads and two synchronized objects.</p>

2.4 Overriding a synchronized method

Name	Overriding a synchronized method.
Description	In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super class, the method in the subclass is said to override method in the super class. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. So, if the overridden method is synchronized and the overriding one is not, the method will be referred to as non synchronized.
Category	Race condition.
Example	<pre> Class MyThread implements Runnable { Ressource r ; public MyThread(Ressource _r) { this.r = _r ; } run() { r.process() ; } } class Ressource { ... synchronized void process() { ... } } class SpecRessource { void process() { ... } } </pre>
Detection	<ol style="list-style-type: none"> 1. Compute the inheritance class 2. Collect the base methods and their overriding ones 3. Identify the synchronized base methods whose overriding ones are not. <p>Detectable by Jlint</p>
Re-Factoring	<p>Solution: Make method synchronized.</p> <p>Conflicts: None</p>
Comments	<p>Source: Http://asktenali.com/Software_Education_e-learning/Java/java.html</p> <p>Not necessarily a bug. So it can be presented as a warning.</p>

2.5 Non synchronized method called by more than one thread

Name	Non synchronized method called by more than one thread
Description	It is not safe when a non synchronized method could be called by more than one thread.
Category	Race condition.
Example	<pre> ... public void nonSyncmethod() { ... } ... class Thread1Calling_NonSync_Method extends Thread{ public void run() { this.obj.nonSyncmethod(); } } class Thread2Calling_NonSync_Method extends Thread{ public void run() { this.obj.nonSyncmethod(); } } </pre>
Detection	<ol style="list-style-type: none"> 1. For each thread, collect the methods called inside the run() method. 2. Extract the methods that could be called by more then one thread. 3. Identify the non-synchronized methods. <p>Detectable by Jlint.</p>
Re-Factoring	<p>Solution: Declare the method synchronized</p> <p>Conflicts: Possibility of oversynchronization</p>
Comments	<p>Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm http://java.quest.com/support/jprobe/docs/JProbeThreadalyzerGuide501.pdf</p> <p>This pattern can be detected dynamically also by either of the following algorithms implemented within Jprobe:</p> <ol style="list-style-type: none"> 1. Happens Before: It seeks to prove that one thread’s access to a variable must happen before another thread can access the same variable. If the analyzer cannot prove a “happens-before” relationship, the threads are deemed to be <i>concurrent</i>, meaning that they may require access to the same data at the same time, and JProbe Threadalyzer reports a data race. 2. Lock Covers: It watches all access to shared variables, and tracks the <i>lock cover</i>—the set of locks held by all threads that ever need access to a shared variable. When the lock cover becomes empty, the variable is deemed to be unprotected and Jprobe Threadalyzer reports the condition as a data race.

2.6 Non volatile field used by more than one thread

Name	Non volatile field used by more than one thread.
Description	<p>Normally, it is not safe to use non volatile variables as shared fields, unless they are synchronized.</p> <p>The Java programming language allows threads that access shared variables to keep private working copies of the variables; this allows a more efficient implementation of multiple threads. But could be source of inconsistency with the master copies in the shared main memory.</p> <p>With volatile modifier, at each update of a variable, Java reflects the changes in the main memory.</p>
Category	Race condition.
Example	
Detection	<ol style="list-style-type: none">1. Collect the attributes used by each thread.2. Detect the non volatile attributes that are shared. <p>Detectable by Jlint</p>
Re-Factoring	<p>Solution: Make the attribute volatile.</p> <p>Conflicts: If volatile fields are accessed frequently inside methods, their use is likely to lead to slower performance than would locking the entire methods. The use of volatile variable is often considered unsafe and could result in dataraces.</p>
Comments	Source: Doug Lea, "Concurrent Programming in Java™ Design principles and patterns". Addison Wesley, 2000.

2.7 Non synchronized run() method

Name	Non synchronized run() method
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	<pre>Class MyThread implements runnable{ ... public void run() {...} } ... MyThread t1 = new MyThread() ; New Thread(t1).start() ; New Thread(t1).start() ;</pre>
Detection	<p>Detect the creation of Java threads (Thread Class) with the same Runnable object, an instance of a class that implements the Runnable interface.</p> <p>Data flow analysis could be used to determine which threads are created with the same Runnable instance.</p> <p>Detectable by Jlint.</p>
Re-Factoring	<p>Solution: Make the run() method synchronized.</p> <p>Conflicts: Oversynchronization.</p>
Comments	Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm

2.8 Overuse of synchronized methods

Name	Overuse of synchronized methods
Description	<p>Methods of a class that is used by only one thread should not be synchronized. Otherwise, it could have a negative impact on the efficiency of the application.</p> <p>Synchronized methods take longer to execute than unsynchronized methods and limit the multitasking present in the JVM and operating system.</p>
Category	Efficiency problem.
Example	
Detection	<ol style="list-style-type: none">1. Collect methods that are called in each thread.2. Detect synchronized methods that are called by only one thread.
Re-Factoring	<p>Solution: Declare the methods non-synchronized</p> <p>Conflicts: If not properly handled, the solution could lead to methods being declared non-synchronized while they should be.</p>
Comments	<p>Source:</p> <p>Http://www-1.ibm.com/servers/eserver/zseries/software/java/perfarchive.html#synch</p> <p>In some cases, performance improvements are achieved by writing synchronized and non-synchronized versions of the same method, and invoking the synchronized method only when needed.</p>

2.9 Method wait() invoked with another object locked

Name	Method <code>wait()</code> invoked with another object locked
Description	<p>This antipattern appears when one thread invokes the <code>wait()</code> method while it is locking other objects.</p> <p>When the <code>wait()</code> method is invoked, more than one monitor objects are locked by the thread. Since <code>wait()</code> unlocks only one monitor, it can be a source of deadlock.</p>
Category	Deadlock.
Example	<pre> Class Resource { ... synchronized process() { ... while() { ... wait(); } } } Thread: ... Resource res ; ... run() { Synchronized(A) { ... res.process() ; } } </pre>
Detection	<ol style="list-style-type: none"> 1. Detect the <code>wait()</code> occurrences in the code of a thread 2. Check the monitors held by the thread at the time <p>Detectable by Jlint</p>
Re-Factoring	<p>Solution: When <code>wait()</code> is invoked in a thread make sure that no more than one lock is withheld</p> <p>Conflicts: Improper handling of the locks could lead to data races and corruption.</p>
Comments	<p>Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm</p> <p>A variation of this antipattern is also cited in the literature. Lock and I/O wait: it occurs when a thread enters a monitor and then waits on a blocking input/output operation. If the blocking I/O operation never occurs, any thread requiring the held monitor blocks indefinitely, causing a deadlock.</p>

2.10 Call sequence to method potentially causing deadlock in wait()

Name	Call sequence can cause deadlock in <code>wait()</code> .
Description	The problem appears when a sequence of method calls, which lock at least two object monitors, terminates by calling the method <code>wait()</code> . Since <code>wait()</code> unlocks only one monitor and suspends the thread in which the call was made, this can cause deadlock.
Category	Deadlock.
Example	
Detection	<ol style="list-style-type: none">1. Compute the call graph2. Analyze the method invocations in the graph.3. Collect the locked objects until the <code>wait()</code> method call is reached. <p>Detectable by Jlint</p>
Re-Factoring	<p>Solution: When <code>wait()</code> is invoked in a thread make sure that no more than one lock is withheld</p> <p>Conflicts: Improper handling of the locks could lead to data races and corruption.</p>
Comments	<p>Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm</p> <p>Similar to: Method <code>wait()</code> invoked with monitor of other object lock, the difference is in detecting the sequence of method calls.</p>

2.11 *identifier.wait()* method called without synchronizing on *identifier*

Name	<i>Identifier.wait()</i> method called without synchronizing on <i>identifier</i> .
Description	<p>One of the constraints in Java is that an <i>identifier.wait()</i> (or <i>identifier.notify()</i>) can only be invoked by a thread currently holding the monitor lock for <i>identifier</i>.</p> <p>When a thread violates this constraint, the thread will be permanently blocked (i.e. deadlocked).</p>
Category	<p>Quality and style problem.</p> <p>Deadlock.</p>
Example	
Detection	<p>Use points-to analysis, data and control flow analysis, call graph.</p> <p>Detectable by Jlint</p>
Re-Factoring	<p>Solution: The thread should get a lock on the object (referenced by <i>identifier</i>) before calling <i>wait()</i>, i.e., the <i>wait()</i> method should only be invoked from inside a <i>synchronized(identifier)</i> block..</p> <p>Conflict: None.</p>
Comments	<p>Source: http://www.cs.kent.ac.uk/projects/ofa/java-threads/203.html</p> <p>The thread could get the monitor in another method that calls the method with the <i>wait()</i> invocation.</p>

2.12 Synchronized read only methods

Name	Synchronized read only methods
Description	<p>Methods that perform just read access on an object (while there are no methods that have a write access) need not be synchronized. They simply add to the overhead of synchronization.</p> <p>It is useful to know how many such cases exist in an application. This serves as an indicator of the level of needed synchronization.</p>
Category	Efficiency problem.
Example	
Detection	Use control and data flow analysis.
Re-Factoring	<p>Solution: Remove synchronization from read only methods.</p> <p>Conflicts: Risk of data races, especially when the fields in the methods are shared among several threads.</p>
Comments	Source: Alan Holub, "Taming Java Threads", Springer-Verlag, 2000.

2.13 Internal call of a method

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	<pre> Class Reentrant { Synchronized void foo() { This.bar() ; } ... synchronized void bar() { } } </pre>
Detection	Use call graph and data flow analysis.
Re-Factoring	<p>Potential Solutions:</p> <ol style="list-style-type: none"> 1. Remove synchronization from the method using the inner lock (bar () in the example). 2. Remove the code from bar () method into the calling method. <p>Conflict: Risk of data races or corruption, especially if other threads call the method.</p>
Comments	<p>Source:</p> <p>Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, Susan J. Eggers: Static Analyses for Eliminating Unnecessary Synchronization from Java Programs.</p> <p>Agostino Cortesi, Gilberto Filé (Eds.): Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22–24, 1999, Proceedings. LNCS 1694 Springer 1999: 19-38.</p>

2.14 Object locked but not used

Name	Object locked but not used
Description	This antipattern is exhibited when a thread gets a monitor on an object and does not use it.
Category	Efficiency problem. Quality and style problem.
Example	
Detection	In each synchronized block, detect if the synchronized object is used in this lock by checking all the statements of the block.
Re-Factoring	Solution: Rewrite the synchronized block Conflicts: Care should be taken not to leave any shared objects unsynchronized.
Comments	Source: Alan Holub "Taming Java Threads", Springer-Verlag, 2000 Limitation: not detected if the object is not represented by an explicit reference as in the following example: <code>synchronized(obj.field){...}</code>

2.15 Synchronization abuse

Name	Synchronization abuse
Description	Do not synchronize an entire method if the method contains significant operations that do not need synchronization. To maximize concurrency in a program, try to minimize the frequency and duration of lock acquisition. Sometimes, only a few operations within a method may require synchronization.
Category	Efficiency problem.. Quality and style problem.
Example	<pre>Synchronized void f() { statement1 ; statement2 ; statement3 ; // synchronization not needed. }</pre>
Detection	Use data flow analysis to detect code that does not require synchronization.
Re-Factoring	Solution: Use of synchronization on a part of the method. Re-factoring the example: <pre>Void f() { Synchronized(this) { Statement1; Statement2 ; } statement3 ; }</pre> Conflicts: Risk of leaving pieces of code unprotected.
Comments	Source: http://www.xp.co.nz/Coding_Standards_for_Java.htm It might not be straightforward to detect parts of code that do not require synchronization. Synchronized blocks are often slightly more expensive than synchronized methods (especially in lock-release).

2.16 wait() not in loop

Name	wait() is not in 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	<pre>if (! resource.isAvailable) { wait() ; }</pre> <p>When the thread wakes up, it is better to check again the condition.</p>
Detection	Check if the wait() method invocation is in a loop. Detectable by FindBugs.
Re-Factoring	Solution: Use while instead of if Conflicts: None
Comments	Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", http://www.cs.umd.edu/~pugh/java/bugs/

2.17 Unconditional wait()

Name	Unconditional <code>wait()</code>
Description	<p>This could be seen as a strange way to implement a wait.</p> <p>When waiting on a monitor, it is good practice to check the condition being waited for before entering the <code>wait()</code>. Without this check, the possibility that the event notification has already occurred is not excluded, and the thread may wait forever.</p>
Category	Deadlock.
Example	<pre>... synchronized(o) { System.out.println ("waiting...") ; wait() ; }</pre>
Detection	<p>In each synchronized block, check if the <code>wait()</code> method call is in a conditional structure.</p> <p>Detectable by FindBugs</p>
Re-Factoring	<p>Solution: always use <code>wait()</code> within a conditional structure.</p> <p>A better version of the example</p> <pre>if (! o.isAvailable()) { System.out.println ("Not available") ; synchronized(o) { System.out.println ("waiting...") ; wait() ; } }</pre> <p>Conflicts: None</p>
Comments	<p>Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy",</p> <p>http://www.cs.umd.edu/~pugh/java/bugs/</p> <p>Similar to: <code>wait()</code> not in loop</p>

2.18 Unconditional notify() or notifyAll()

Name	Unconditional <code>notify()</code> or <code>notifyAll()</code>
Description	This could be seen as the dual of unconditional <code>wait()</code> pattern. In general, because some conditions become true, the <code>notify()</code> and <code>notifyAll()</code> methods are called.
Category	Deadlock. Problem with unpredictable consequences.
Example	
Detection	Check for each synchronized block, if <code>notify()</code> (or <code>notifyAll()</code>) is called within a conditional structure. Detectable by FindBugs
Re-Factoring	Solution: Always make sure that <code>notify()</code> and <code>notifyAll()</code> are called from within conditional structures. Conflicts: None.
Comments	Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", http://www.cs.umd.edu/~pugh/java/bugs/

2.19 Reference value is changed when it is used in synchronization block

Name	Reference value is changed when it is used in synchronization block.
Description	This can be seen as a programming error.
Category	Quality and style problem. Deadlock.
Example	<pre>Synchronized(ref) { ... ref = new Object() ; ... }</pre>
Detection	Detect the instructions that change the reference in a synchronized block. Detectable by Jlint.
Re-Factoring	Not available.
Comments	Source: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm

2.20 Overthreading

Name	Overthreading
Description	It is exhibited when a “large” number of threads are defined and created.
Category	Efficiency problem. Quality and style problem.
Example	
Detection	Use some indicators, defined statically or dynamically such as: <ul style="list-style-type: none">• No or few objects are shared by threads• No use of any inter-thread synchronization: <code>wait</code>, <code>notify</code>• Size of threads (Lines of code-LOC- metric)• Thread execution is sequential
Re-Factoring	Solution: Use threading only as needed. Conflict: Risk of ending with blob threads.
Comments	Source: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreadsthreading.asp

2.21 Blob Thread

Name	Blob thread AKA, God thread
Description	A thread that takes the whole or a large part of the activity of the system.
Category	Quality and style problem.
Example	
Detection	Use dynamic and static analysis to collect indicators of: <ul style="list-style-type: none">• Inter thread activities (sharing objects, wait and notify methods)• Size of the thread (LOC metric).• Activity of each thread.
Re-Factoring	Solution: Distribute program activities among threads Conflict: Overthreading
Comments	Source: Connie U. Smith and Lloyd G. Williams: <i>Software performance antipatterns</i> . Workshop on Software and Performance, 2000, pp.127-136.

2.22 Complex computation within an AWT/Swing thread

Name	Complex computation within an AWT/Swing thread
Description	The listeners in a Java's Abstract Windows Interface (AWT) thread are too long (temporary irresponsible interface), never end, e.g., due to a cycle with a continuous true condition (irresponsiveness) share many objects with the main thread (may affect responsiveness)
Category	Efficiency problem.
Example	see Allen Holub "Taming Java Threads" [Hol00]: Listing 1.1
Detection	<ol style="list-style-type: none">1. For the AWT thread, check the complexity of the listener method (method names which contain word listen).2. List all the overcomplicated listeners of AWT thread <p>Syntactically, the complexity could be estimated with a metric on the size and number of loops.</p>
Re-Factoring	<p>Solution: Create additional threads to perform any non-trivial operations. Provide the user with a message that a certain operation is in process and with the "Cancel" button.</p> <p>Conflicts: Possibility of overthreading.</p>
Comments	<p>Source: Alan Holub, "Taming Java Threads", Springer-Verlag, 2000</p> <p>The pattern is important since it is relevant to all non-console applications, and interface is the only justified place for threads in a program of a beginner.</p>

2.23 Misuse of notifyAll()

Name	Misuse of <code>notifyAll()</code>
Description	The <code>notify()</code> method is more efficient than the <code>notifyAll()</code> method, especially when not too many objects are shared between threads.
Category	Efficiency problem.
Example	
Detection	<ol style="list-style-type: none">1. Collect all objects used in each thread.2. Search for objects that are used by only two threads.
Re-Factoring	<p>Solution: Replace <code>notifyAll()</code> by <code>notify()</code></p> <p>Conflicts: Since <code>notify()</code> activates one thread only (randomly) this fix could lead to more overhead and possibly a deadlock.</p>
Comments	<p>Source:</p> <p>http://javafaq.nu/article211.html</p> <p>Generally, the use of <code>notifyAll()</code> is more safe for multithreaded programming.</p> <p>When <code>notifyAll()</code> is called, all threads waiting for the same object-lock, will receive a signal but just one gets the lock. Therefore, when there are only two threads operating on this object, the use of <code>notifyAll()</code> is not needed.</p>

2.24 The double-check locking for synchronized initialization

Name	The double-check locking for synchronized initialization
Description	Double-check locking is widely cited and used as an efficient method for implementing lazy initialization in a multithreaded environment and as an efficient way to reduce synchronization overhead. Unfortunately, it will not work reliably in a platform independent way when implemented in Java, namely due to the current Java Memory Model (JMM).
Category	Race condition.
Example	<pre>//Load and initialize our instance //if necessary if (instance == null) { synchronized (this) { if (instance == null) ... create the instance ...</pre>
Detection	<p>For basic instances of the antipattern, like the one illustrated in the example, detect code segments that match exactly the listing.</p> <p>Detectable by FindBugs.</p>
Re-Factoring	<p>Solution: Two possible fixes</p> <ol style="list-style-type: none">1. Use a temporary variable to try and force the constructor to execute before its reference is assigned.2. Use a "guard" variable to indicate that the initialization has completed <p>Conflicts: Unexpected behavior, as there is no guarantee the compiler or the JVM will follow the ordering from the source code.</p>
Comments	<p>Source: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html</p> <p>The Java Language Specification (JLS) gives Java compilers and JVMs a good deal of latitude to reorder or optimize away operations. Java compilers are only required to maintain within-thread as-if-serial semantics, which means that the executing thread must not be able to detect any of these optimizations or reorderings. However, the JLS makes it clear that in the absence of synchronization, other threads might perceive memory updates in an order that "may be surprising."</p>

2.25 Synchronized atomic operations

Name	Synchronized atomic operations
Description	An atomic operation is one that cannot be interrupted by the thread scheduler—if the operation begins, it will run to completion before the possibility of a context switch, it is the case of primitive type variables except double and long types. <code>Get()</code> and <code>Set()</code> methods could be used without synchronization when considered as atomic. This antipattern is highly disputed.
Category	Efficiency problem.
Example	
Detection	Detect in the code all the atomic operations inside synchronized blocks or methods.
Re-Factoring	<p>Solution: Do not synchronize atomic operations.</p> <p>Conflicts: Have to be careful about what statements are really atomic.</p> <p>Data incoherence: without synchronization the values assigned by one thread are not visible by other threads.</p>
Comments	<p>Source:</p> <p>Alan Holub, "Taming Java Threads", Springer-Verlag, 2000.</p> <p>Some would argue that atomic operations still need synchronization.</p>

2.26 Synchronized immutable object

Name	Synchronized immutable object
Description	An immutable object is one whose state does not change after it is created (fields are final). Multiple threads can safely access the object simultaneously, so no synchronization is required.
Category	Efficiency problem.
Example	
Detection	For every reference used as a synchronized object in a synchronized block, check if the reference is declared final.
Re-Factoring	<p>Solution:</p> <p>Remove synchronization from pieces of code handling immutable objects.</p> <p>Conflicts: None.</p>
Comments	<p>Source:</p> <p>Alan Holub, "Taming Java Threads", Springer-Verlag, 2000.</p> <p>When the synchronization is not defined by synchronized block (synchronized methods are used instead), deeper analysis should be performed: All instances of the class (with synchronized methods) must be used with final fields.</p> <p>Points-to analysis can improve results, when a final field is referenced by another variable.</p>

2.27 Unnecessary notification

Name	Unnecessary notification
Description	Notification issued when no thread is waiting.
Category	Efficiency problem. Quality and style problem.
Example	
Detection	For each thread that may call <code>notify()</code> or <code>notifyAll()</code> methods, check if another thread could call <code>wait()</code> method on the same object.
Re-Factoring	Solution: Remove notification operations when no threads are waiting. Conflicts: Some threads might end up waiting forever.
Comments	Source: http://www.python.org/doc/current/lib/condition-objects.html

2.28 Double call of the start method of a thread

Name	Double call of the <code>start()</code> method of a thread
Description	The <code>start()</code> method call is not supposed to be used more than once for the same thread.
Category	Problem with unpredictable consequences.
Example	<pre>... t = new Thread(runnable) ; t.start() ; ... t.start() ;</pre>
Detection	<p>For each creation of an instance of a thread (Thread Class), check if there are more than one call to <code>start()</code> method.</p> <p>Sometimes, the detection of the second invocation needs data flow analysis or a points-to analysis.</p>
Re-Factoring	<p>Solution: Remove the additional unneeded <code>start()</code>.</p> <p>Conflicts: None.</p>
Comments	<p>Source: Patrick Naughton, "The Java Handbook", McGrawHill, 1996.</p> <p>The second invocation of the <code>start()</code> method, could be far from the first one, possibly in another method, or "via" an alias; therefore, making detection quite complex:</p> <pre>t = new thread() ; t.start() ; ... tt = t ; ... tt.start() ;</pre>

2.29 Waiting forever

Name	Waiting forever
Description	Thread executes the <code>wait()</code> for the lock of an object but is never notified and thus never resumes its execution. It could be considered as a programming error.
Category	Deadlock.
Example	
Detection	For each thread that could call the <code>wait()</code> method, check if there is at least, one thread that may call <code>notify()</code> or <code>notifyAll()</code> methods for the same object.
Re-Factoring	Solution: Make sure all instances of <code>wait()</code> are notified. Conflicts: Risk of excessive unnecessary notification.
Comments	Source: http://java.quest.com/support/jprobe/docs/JprobeThreadalyzerGuide501.pdf A variation of this antipattern may appear as the Wait Stall antipattern

2.30 Unsynchronized spin-wait

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	Efficiency problem. Livelock
Example	<pre>//wait for spider to finish while (spider.isAlive()) { //do nothing, just test again }</pre>
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. Detectable by FindBugs.
Re-Factoring	Solution: (For simple cases as in the example) Use <code>yield</code> or <code>sleep</code> 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 facilities could lead to deadlocks or data races.
Comments	Source: D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", http://www.cs.umd.edu/~pugh/java/bugs/ In general, this is hard to detect this antipattern.

2.31 Start() method call in constructor

Name	Start () method call in constructor
Description	<p>start () method call in constructor of a class that implements Runnable.</p> <p>In the case where this class is specialized (defining a subclass), the thread is launched (invocation of the start ()) before the end of the instance creation of the subclass, because the constructor of the super class is executed before the one of the current class.</p> <p>The program can have an unexpected behavior.</p>
Category	Problem with unpredictable consequences.
Example	
Detection	Detectable by FindBugs
Re-Factoring	<p>Solution: Rewrite the constructor method and avoid calling start () in it, or declare the class final.</p> <p>Conflicts: None.</p>
Comments	<p>Source: Findbugs.</p> <p>Similar to: Double call of the start method of a thread.</p>

2.32 Get-Set methods with different declarations

Name	Get-Set methods with different declarations
Description	This antipattern occurs when the get method of an object is unsynchronized while the set method is synchronized.
Category	Race condition.
Example	
Detection	Detectable by FindBugs
Re-Factoring	Solution: Declare both methods synchronized. Conflicts: Possibility of over synchronization.
Comments	Source: Findbugs.

2.33 Improper method calls

Name	Improper method calls
Description	This antipattern occurs when improper method calls are made in the methods of a thread. A method call can be improper if: The call itself is not correct (<code>Thread.run()</code>). The called method has bugs. The method is deprecated.
Category	Problem with unpredictable consequences.
Example	
Detection	Detectable by FindBugs
Re-Factoring	Solution: Rewrite the method in which the improper call is made. Conflicts: None.
Comments	Source: Findbugs.

2.34 Wait stall

Name	Wait stall
Description	A wait stall can occur when a thread calls a <code>wait()</code> method with no timeout specified.
Category	Livelock. Deadlock.
Example	
Detection	Detectable dynamically by Jprobe Threadalyzer. The Jprobe Threadalyzer sets a priori a threshold for blocked threads. If the <code>wait()</code> method does not complete before the threshold is reached, then the stall will be reported as a wait stall.
Re-Factoring	Solution: Use <code>wait()</code> with timeout. Conflicts: It could affect the efficiency.
Comments	Source: http://java.quest.com/support/jprobe/docs/JprobeThreadalyzerGuide501.pdf

2.35 Premature `join()` call

Name	Premature <code>join()</code> call
Description	A call to the <code>join()</code> method of a thread is premature if this thread has not been started at the time of the call. In Java such calls are simply ignored, but their presence is alarming because they may indicate a fault in the program logic or non-optimal code.
Category	Quality and style problem.
Example	
Detection	A data flow analysis is needed, but dynamic analysis could be more efficient. Detectable by Flavers.
Re-Factoring	Solution: Rethink the logic of the program to make sure that the <code>join()</code> method of a thread is not called until it is already started. Conflicts: None.
Comments	Source: http://cis.poly.edu/gnaumovi/papers/flavers-java.pdf

2.36 Dead interactions

Name	Dead interactions
Description	The antipattern appears when a thread calls another thread after the target thread has already terminated.
Category	Quality and style problem
Example	
Detection	<p>A data flow analysis is needed, but dynamic analysis could be more efficient.</p> <p>Detectable by Flavers.</p>
Re-Factoring	<p>Solution: Rethink the logic of the program to make sure that the call to a thread is not made after the thread has terminated.</p> <p>Conflicts: None.</p>
Comments	<p>Source: http://cis.poly.edu/gnaumovi/papers/flavers-java.pdf</p>

2.37 Calling `Join()` for an immortal thread

Name	Calling <code>join()</code> for an immortal thread
Description	The problem occurs when <code>join()</code> is called for a thread that would not terminate such as a daemon or the main thread.
Category	Efficiency problem.
Example	
Detection	A data flow analysis is needed, but dynamic analysis could be more efficient. Detectable by Flavers.
Re-Factoring	Solution: Rethink the logic of the program to make sure that <code>join()</code> is called only for threads that would terminate. Conflicts: None.
Comments	Source: http://cis.poly.edu/gnaumovi/papers/flavers-java.pdf

REFERENCES

- [AH03] Allen Goldberg and Klaus Havelund. Instrumentation of Java Bytecode for Runtime Analysis. Fifth ECOOP Workshop on Formal Techniques for Java-like Programs, Darmstadt, Germany, July 21, 2003.
- [Ald99] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, Susan J. Eggers: Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. Agostino Cortesi, Gilberto Filé (Eds.): Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22–24, 1999, Proceedings. LNCS 1694 Springer 1999: 19-38.
- [All01] Eric E. Allen: Bug patterns: An introduction Developer Works [online] <http://www-106.ibm.com/developerworks> February 1, 2001.
- [AntiPatterns] The antipattern, the book. <http://www.antipatterns.com/thebook.htm>
- [Arm98] Eric Armstrong. Hotspot: A new breed of virtual machine, 1998. <http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html>
- [Arn01] Matthew Arnold, Barbara G. Ryder A Framework for Reducing the Cost of Instrumented Code SIGPLAN Conference on Programming Language Design and Implementation. 2001.
- [Art01] Cyrille Artho: Finding faults in multi-threaded programs. Master Thesis. Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin. 2001.
- [Art03] Cyrille Artho, Klaus Havelund, and Armin Biere: High-level Data Races. The First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03), Angers, France, April 2003.
- [AS85] Bowen Alpern, Fred B. Schneider: Defining Liveness. Information Processing Letters, October 1985.
- [Bac96] D.F Bacon and P.F. Sweeney. *Fast Static Analysis of C++ Virtual Function Calls*. In OOPSLA'96, pages 324-341, November 1996. ACM Press.
- [Bau03] 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 9, 2003.
- [Ber97] P. Bertelsen. Semantics of Java bytecode. A Technical Report. Department of mathematics and physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997. <http://www.dina.kvl.dk/~pmb>
- [Bitter] Bruce Tate: Bitter Java Web Site. [website] <http://www.j2life.com/bitterjava> .
- [BJS] Barcelona Java Suite (BJS) <http://www.cepba.ucp.es/BJS>

[Blo02] Joshua Bloch, *Effective Java: Programming Language Guide*. Addison Wesley, 2002.

[Bod03] Eric Bodden. *JAnalyzer: A Visual Static Analyzer for Java*.
<http://janalyzer.bodden.de>

[Bog00] Jeffrey Bogda: Detecting Read-Only Methods in Java. Sandhya Dwarkadas (Ed.): *Languages, Compilers, and Run-Time Systems for Scalable Computers*, 5th International Workshop, LCR 2000, Rochester, NY, USA, May 25-27, 2000, Selected Papers. *Lecture Notes in Computer Science 1915* Springer 2000: 143-154.

[Bog01]] Jeffrey Bogda: *Program Analysis Alleviates Java Synchronization*. Ph.D. Dissertation. University of California. 2001.

[Bog99]] Jeffrey Bogda and Urs Hölzle: Removing Unnecessary Synchronization in Java. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*, Denver, Colorado, USA, November 1–5, 1999. *SIGPLAN Notices* 34(10), October 1999:35-46.

[Boy01] Chandrasekhar Boyapati, Martin C. Rinard: A Parameterized Type System for Race-Free Java Programs. *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, October 14–18, 2001, Tampa, Florida, USA. *SIGPLAN Notices* 36(11), November 2001, ACM: 56-69.

[Bra00] Guillaume Brat, Seungjoon Park, Klaus Havelund, and Willem Visser: *Java PathFinder – Second Generation of a Java Model Checker*. *Proc. of Post-CAV Workshop on Advances in Verification*, Chicago, July 2000.

[Bra01] Guillaume Brat and Willem Visser: Combining Static Analysis and Model Checking for Software Analysis. Feather, M. ,Goedicke, M. (Eds.) *Proc. of the 16th International Conference on Automated Software Engineering (ASE'01)*. Coronado Island, California, November 26-29, 2001 IEEE Computer Society, November 2001.

[Bro02] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge and Qin Wang *STEP: A Framework for the Efficient Encoding of General Trace Data* back Date: November 2002 *PASTE 2002*, Charleston, SC, USA.

[Caffeine] Y.-G. Gueheneuc, Caffeine.
<http://www.yann-gael.gueheneuc.net/Work/Research/Caffeine>

[Cai03] Andrew Cain, Jean-Guy Schneider, Doug Grant and Tsong Yueh Chen. *Run-time Data Analysis for Java Programs*. 1st Workshop on ASARTI, Darmstadt, Germany, July 21, 2003.

[Car02] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. *An Instrumentation Tool for Threaded Java Application Servers*. XIII Jornadas de Paralelismo, Lleida, Spain, September 2002.

[Car03] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. Complete Instrumentation Requirement for Performance Analysis of Web based Technologies. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), Austin, Texas, USA, March 2003.

[Cha96] Albert Tymothy Chamillard, Lori A. Clarke, and George S. Avrunin: An empirical comparison of static concurrency analysis techniques. Technical Report 96-84, Department of Computer Science, University of Massachusetts, 1996. Revised May 1997.

[Cho01] J.-D. Choi, A. Loginov, and V. Sarkar: Static Datarace Analysis for Multithreaded Object-Oriented Programs. Technical report, IBM Research, 2001. Report RC22146.

[Cho02] J.-D. Choi and A. Zeller Isolating Failure-Inducing Thread Schedules International Symposium on Software Testing and Analysis (ISSTA2002), Via di Ripetta, Rome - Italy, July 22-24, 2002.

[Cho02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar and Manu Sridharan: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02), June 2002.

[Cho98] Jong-Deok Choi, Harini Srinivasan: Deterministic Replay of Java Multithreaded Applications. ACM SIGMETRICS Symposium on Parallel and Distributed TOOLS (SPDT). August 1998.

[Cho99] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, Samuel P. Midkiff: Escape Analysis for Java. Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999. SIGPLAN Notices 34(10), October 1999: 1-19.

[Cob02] Jamieson M. Cobleigh, Lori A. Clarke, Leon J. Osterweil: FLAVERS: a Finite State Verification Technique for Software Systems. IBM Systems Journal on Software Testing and Verification, 41(1), 2002:140-165.

[Coh] S. Cohen. JTrek. Developed by Digital.

[Coh98] G. A. Cohen, D. L. Kaminsky and J. S. Chase. Automatic Program Transformation with JOIE. In Proceedings USENIX Annual Technical Symposium, 1998.

[Cop03] Tom Copelend: Custom PMD Rules. O'Reily. Onjava.com. 04/09/2003. [on-line] <http://www.onjava.com> .

[Cor00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng: Bandera: Extracting Finite-state Models from Java Source Code. International Conference on Software Engineering. 2000: 439-448

- [Cor95] James C. Corbett and George S. Avrunin: Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, January 1995, 6:97–123.
- [Cor98] James C. Corbett: Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. Technical Report ICS-TR-98-20, Department of Information and Computer Science, University of Hawaii, October 14, 1998.
- [Cor99] James C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, vol. 22, No. 3, March 1996: 161-180.
- [Dah01] M. Dahm: *Byte code engineering with the BCEL API* Technical Report B-17-98, Freie Universit at Berlin, Institut fur Informatik, April 2001.
- [Dah99] M Dahm. Byte Code Engineering. In *JIT 99 Proceeding*.
- [DC01] M. Deters and Ron K. Cytron. Introduction of Program Instrumentation Using Aspects. *OOPSLA 2001 Workshop on Advanced Separation of Concerns*, October 2001.
- [Dea95] J. Dean, D. Grove, and C. Chambers. *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*. *Proceedings of ECOOP'95. Lecture Notes in Computer Science*, 952:77–101, 1995.
- [Dem98] C.Demartini and R.Iosif and R.Sisto: Modeling and Validation of Java Multithreading Applications using SPIN. In *Proceedings of the 4th SPIN workshop*, Paris, France, November 1998.
- [DWY] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. "Property Specification Patterns for Finite-state Verification". *2nd Workshop on Formal Methods in Software Practice*, March, 1998.
- [DY] Laura K. Dillon and Qing Yu "Specification and Testing of Temporal Logic Properties of Concurrent System Designs" Technical Report.
- [Ede02] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur: Multithreaded Java program test generation. *IBM Systems Journal*, 41(1) 2002:111-125.
- [Elr01] T. Elrad, R. E. Filman and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, Volume 44 Issue 10, October 2001.
- [Eng00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem: [Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions](#). *4th Symposium on Operating System Design & Implementation*. San Diego, USA. October 22-25, 2000.
- [Far03] Eitan Farchi, [Yarden Nir](#), [Shmuel Ur](#): Concurrent Bug Patterns and How to Test Them. *Proc. of International Parallel and Distributed Processing Symposium*. 2003.

[FH02] R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. Foundations of Aspect-Oriented Languages (FOAL'02), Enschede, Netherlands, April 22, 2002.

[Fla01] Cormac Flanagan, Stephen N. Freund: Detecting Race Conditions in Large Programs. In Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), June 2001.

[Fla02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata: Extended static checking for Java. Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany. Vol.37 Issue 5: ACM Press New York, NY, USA, 2002: 234–245.

[Forum] From: Keith D Gregory (kdg_nospam@world.std.com): Thread question: extend Thread vs implements Runnable (2001, 2 octobre). [A message into a java newsgroup], [online] group://comp.lang.java.

[Gei01] M.C.W. Geilen. “On the construction of monitors for temporal logic properties” RV'01 - First Workshop on Run-time Verification, July 23, 2001 Paris, France.

[Gei98] Jennifer Geis: JavaWizard: Investigating Defect Detection and Analysis. Master's thesis. University of Hawaii. May, 1998.

[GH03] Allen Goldberg and Klaus Havelund. Instrumentation of Java Bytecode for Run-time Analysis. Fifth ECOOP Workshop on Formal Techniques for Java-like Programs, Darmstadt, Germany, July 21, 2003.

[God97] P. Godefroid: Model Checking for Programming Languages using VeriSoft. In Proc. 24th ACM Symposium on Principles of Programming Languages, pages 174–186, France, 1997.

[Gor98] M.J.C. Gordon: HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A.Subrahmanyam, editors, VLSI Specification, Verification and Synthesis, p. 73-128. Kluwer, Dordrecht, The Netherlands, 1998.

[Gos00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. The Java Language Specification. Addison-Wesley, second edition, 2000.

[Gui00a] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Preliminary Experiences using the Java Instrumentation Suite. Research Report number: UPC-DAC-2000-25 / UPC-CEPBA-2000-12, April 2000.

[Gui00b] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. 2nd Workshop on Java for High Performance Computing, Santa Fe, New Mexico, USA, May 7, 2000.

[Gui00c] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Instrumentation Environment for Java Threaded Applications. XI Jornadas de Paralelismo, Granada, Spain. September 2000.

- [Hal03] Hallal, H., Boroday, S., Ulrich, A. and Petrenko, A. "An Automata-based Approach to Property Testing in Event Traces" In Proceedings of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003), pp. 180-196. Sophia Antipolis, France, May 26-29, 2003.
- [Hav00a] K. Havelund and T. Pressburger: Model Checking Java Programs Using Java Pathfinder. International Journal on Software Tools for Technology Transfer, STTT, 2(4), April 2000.
- [Hav00b] K. Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs, Proc. Seventh SPIN Workshop, 2000: 245-264.
- [Hav01] K. Havelund, Scott Johnson and G. Rosu. "Specification and Error Pattern Based Program Monitoring". European Space Agency Workshop on On-Board Autonomy, Noordwijk, Holland, October 2001.
- [Hav01a] Klaus Havelund and Grigore Rosu: *Java PathExplorer – A Runtime Verification Tool* In proceedings of the Sixth International Symposium on AI, Robotics, and Automation in Space, May 2001.
- [Hav01b] Klaus Havelund and Grigore Rosu: *Monitoring Java Programs with Java PathExplorer*. First Workshop on Runtime Verification(RV'01), Paris, France, 23 July 2001.
- [Hav02] Klaus Havelund "Dynamic Program Analysis" A talk at NASA Ames Research Center, October 2002.
- [Hav03] 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". ASM 2003, 10th International Workshop on Abstract State Machines Taormina, Italy, March, 2003.
- [Hav03b] Klaus Havelund and Scott D. Stoller and Shmuel Ur: Benchmark and Framework for Encouraging Research on Multi-Threaded Testing. Invited paper PADTAD'03, Parallel and Distributed Systems: Testing and Debugging Nice, France, April 22-26, 2003.
- [Heu03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, Welf Löwe. Automatic Design Pattern Detection. : 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA. p. 94.
- [Hoa74] C. Hoare: Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10) 1974:549-557.
- [Hol00] Allan Holub: Taming Java Threads. Sun Microsystems Press. 2000.
- [Hov03] David Hovemeyer and William Pugh: Finding Bugs is Easy (submitted).
- [HR] Klaus Havelund, Grigore Rosu. "*Efficient Monitoring of Safety Properties*". Software Tools and Technology Transfer. To appear.

[HR01] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. First Workshop on Run-time Verification (RV'01), Paris, France, 23 July 2001.

[HR02] K. Havelund and G. Rosu “Synthesizing Monitors for Safety Properties” International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France, April 14, 2002.

[Hua79] J. C. Huang. Detection of Data Flow Anomaly Through Program Instrumentation. IEEE Transactions on Software Engineering, Volume 5, January 1979.

[IBM00] IBM AlphaWorks. CFParse, September, 2000
<http://www.alphaworks.ibm.com/tech/cfparse>

[IBM002] IBM AlphaWorks. Jikes ByteCode Toolkit, March 2000
<http://www.alphaworks.ibm.com/tech/jikesbt>

[Javassist] Java Assistant <http://www.csg.is.titech.ac.jp/~chiba/javassist>

[JContractor] Java implementation of Design By Contract for the Java language (online)
<http://jcontractor.sourceforge.net>

[JFluid] Sun JFluid, <http://research.sun.com/projects/jfluid> \

[JLS] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: Sun Java Language Specification. Second Edition. Sun Publishing.

[JPax] Ron LeMaster and David Leberknight. JpaX
<http://ic.arc.nasa.gov/researchinfusion/materials/JPaX/talk.pdf>

[JPDA] Sun Microsystems. Java Platform Debugger Architecture
<http://java.sun.com/j2se/1.4.1/docs/guide/jpda>

[JProbe] Quest. JProbe. <http://www.quest.com/jprobe>

[JRaT] Java Run-time Analysis Tool (online) at <http://jrat.sourceforge.net>

[JVM] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification. Second edition. <http://java.sun.com/docs/books/vmspec>

[JVMPi] Sun Microsystems. Java Virtual Machine Profiling Architecture
<http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>

[Kar98] Murat Karaorman, Urs Holzle and John Bruno. “jContractor: A Reflective Java Library to Support Design By Contract”. A Technical Report, 1998.

[KH98] Ralph Keller and Urs Holzle. Binary Component Adaptation. Proceedings of ECOOP, Brussels, July 1998.

- [Khu02] S. Khurshid, D. Marinov, and D. Jackson: An Analyzable Annotation Language. Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming Systems, Languages and Applications, October 2002.
- [Kim01] Moonjoo Kim. Information extraction for run-time formal analysis. Ph.D Thesis, CIS Department, University of Pennsylvania, September 2001.
- [Kum00] Neel V. Kumar: Multi-threading in Java programs. Developer Works. [on-line] <http://www-106.ibm.com/developerworks/java/library/j-multithreading.html>. March. 2000.
- [Lam77] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". IEEE Transactions on Software Engineering SE-3 March 1977.
- [Lee02a] Jooyong Lee, Ki-Seok Bang, Jin-Young Choi Linkage of Model Checking to Debugger Using Extended JPDA 2002.
- [Lee02b] Jooyong Lee, Ki-Seok and Jin-Young Choi. Systematic testing of Java programs using extended JPDA and reflection. The 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, Nevada, USA, June 24-27, 2002.
- [Lee03] Jooyong Lee, Ki-Seok and Jin-Young Choi. Model Checking in Java Program Debugger. 10th Annual International Static Analysis Symposium(SAS'03), San Diego, California, USA, June 11-13, 2003. (submitted).
- [Lee99] Doug Lee: Concurrent Programming in Java: Design Principles and Patterns by Doug Lea. Second edition. Addison-Wesley, November 1999.
- [Lev00] 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), 2000: 26-38.
- [Lew03] Bil Lewis "Debugging Backward in Time" Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), September 2003.
- [Log4j] Log4j Project <http://jakarta.apache.org/log4j>
- [Lon03] Brad Long, Daniel Hoffman, Paul A. Strooper: Tool Support for Testing Concurrent Java Components. Transactions on Software Engineering 29(6) June 2003: 555-566.
- [LZ97] H. B. Lee and B. G. Zorn. BIT: A tool for Instrumenting Java bytecodes. In USENIX Symposium on Internet Technologies and Systems, 1998.
- [Mag99] J. Magee and J. Kramer: Concurrency: State Models & Java Programs: John Wiley & Sons, 1999.
- [Mel03] Rob van der Mei, Marcel Harkema, Dick Quartel, and Bart Gijzen. JPMT: a Java Performance Monitoring Tool. To appear in Proceedings TOOLS-2003 (Urbana, USA).
- [MemModel] Vishnu Kotrajaras: Formal study: Java Memory Model and Concurrency. (on-line) <http://www.doc.ic.ac.uk/~vk1/webMemModel>.

- [Mye97] G.J Myers. The Art of Software Testing. John Wiley and Sons, 1978.
- [Nau99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke: Data flow analysis for checking properties of concurrent Java programs. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, May 1999:399-410.
- [Net90] R. Netzer and B. Miller: Detecting data races in parallel program executions. In Advances in Languages and Compilers for Parallel Computing Workshop, , Irvine, Calif., 1990. Cambridge, Mass.: MIT Press 1990:109-129.
- [ODB] Bil Lewis “Omniscient Debugger (ODB)” (online)
<http://www.lamdacs.com/debugger/ODBDescription.html>
- [Owr96] S. Owre, S.Rajan, J. Rushby, N.Shankar, and M.Srivas: PVS : Combining Specification, Proof Checking, and Model Cheking. In R.Alur and T.A.Henzinger, editors, Computer-Aided Verification, CAV'96, number 1102 in LNCS, New Brunswick, NJ, July/August 1996. Springer-Verlag 1996:411-414.
- [Par00] David Y.W. Park, Ulrich Stern, David L. Dill: Java Model Checking. First International Workshop on Automated Program Analysis, Testing, and Verification, 2000.
- [Practices] Java Practices <http://www.javapractices.com>.
- [Prose] The Prose site <http://prose.ethz.ch>
- [PU03] Pietschker, A. Ulrich, A. A Light-weight Method for Trace Analysis to Support Fault Diagnosis in Concurrent Systems. Journal of Systemics, Cybernetics and Informatics, Vol 1. no 2. 2003.
- [Pug00] William Pugh: *The Java Memory Model is Fatally Flawed*. Concurrency: Practice and Experience", volume 12, number 6, pages 445- 455, May 2000.
- [PVM] PVM Java source analyzer. [on-line] <http://pvm.sourceforge.com>.
- [PW02] Paul Pazandak and David Wells. ProbeMeister: Distributed Run-time Software Instrumentation. USE 2002 Spain, June 2002.
- [RC03] Grigore Rosu and Feng Chen. “*Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*”. Third International Workshop on Run-time Verification (RV'03). Boulder, Colorado, U.S.A, July 13, 2003.
- [Ref] Sun Microsystems. Reflection. <http://java.sun.com/j2se/1.4/docs/guide/reflection>
- [Rin01] Martin Rinard: Analysis of Multithreaded Programs. Laboratory for Computer Science, MIT. Lecture Notes in Computer Science. In Proceedings of 8th Static Analysis Symposium, Paris, France, July 2001.
- [RW02] Grigore Rosu, Jonathan Whittle. “*Towards Certifying Domain-Specific Properties of Synthesized Code*”. Verification and Computational Logic (VCL'02), Pittsburgh, PA, USA, 5 October 2002.

- [SAL] The SAL Group: The SAL intermediate Language. Technical report, UC Berkeley, SRI, Stanford University, 1999.
- [Sal01] A. Salcianu: Pointer analysis and its applications for Java programs. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. 2001.
- [San02] Antonella Santone, Gigliola Vaglini: A Tableau-Based Procedure for Model Checking Programs. COMPSAC 2002: 723-730.
- [Sav97] 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. 15(4): 391-411, 1997.
- [SB99] Michael Snyder and Jim Blandy. Heisenberg Debugging Technology. <http://sources.redhat.com/gdb/talks/esc-west-1999/INTROSPECT.html>
- [SDK] Java™ 2 SDK Documentation, Standard Edition [online] www.sun.com
- [Sen03] Koushik Sen, Grigore Rosu and Gul Agha: *Runtime Safety Analysis of Multithreaded Programs*, Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.
- [Shi00] Jack Shirazi: Java Performance Tuning. O'Reily. September 2000.
- [Smi00] Connie U. Smith and Lloyd G. Williams: *Software performance antipatterns*. Workshop on Software and Performance, 2000, pp.127-136.
- [Sno88] R. Snodgrass. A relational approach to monitoring complex systems. In ACM Transaction on Computer Systems, May 1988.
- [Spin] LTL model-checking with SPIN [on-line] <http://www.spinroot.com>.
- [SSP] Software Surveyor Project (online) <http://www.objs.com/DASADA/index.html>
- [Sti98] G.S. Stiles: Safe and Verifiable Design of Multithreaded Java Programs with CSP and FDR. Proc. Workshop on Formal Underpinnings of Java, OOPSLA '98, Oct. 1998. Springer LNCS.
- [Sto02a] Scott D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. International Journal on Software Tools for Technology Transfer, October 2002. Springer-Verlag 4(1): 71-91.
- [Sto02b] S. D. Stoller: Testing concurrent java programs using randomized scheduling. In In Proceedings of the Second Workshop on Runtime Verification (RV), volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.
- [Sun00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vall'ee-Rai, P. Lam, E. Gagnon, and C. Godin. *Practical virtual method call resolution for Java*. In Proceedings of OOPSLA'00, pages 264–280,2000. ACM Press.

[Trace] org.jmonde.debug.Trace <http://www.geocities.com/mcphailmj/Trace>

[Tutorial] Java Tutorial. [on-line] <http://www.sun.com>.

[Ulr03] Ulrich, A., Hallal, H., Petrenko, A. and Boroday, S. "Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis" In Proceedings of the IEEE 36th Hawaii International Conference on System Sciences (HICSS-36), Hawaii, USA, January 6-9, 2003. [CD-Rom].

[Val98] R. Vallee-Rai and L.J. Hendren: Jimple: Simplifying Java Byte code for Analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[Verisoft] "A tool for systematic software testing by Bell Laboratories" Documentation (online) at <http://cm.bell-labs.com/who/god/verisoft>

[Vis03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park and Flavio Lerda: Model Checking Programs. Automated Software Engineering. 10 (2) April 2003: 203–233.

[Wiki] Antipatterns. <http://c2.com/cgi/wiki?AntiPattern>

[Wiki2] Portland Pattern Repository's Wiki website. [on-line] <http://c2.com/cgi/wiki>.

[Yan98] A. S. Cheer-Sun Yang and L. Pollock: All-du-path coverage for parallel programs. ACM SigSoft International Symposium on Software Testing and Analysis, March 1998 23(2):153–162.

[You88] Michal Young, Richard N. Taylor: Combining Static Concurrency Analysis with Symbolic Execution. TSE 14(10) 1988: 1499-1511.

[Zen03] Fancong Zeng: Deadlock Resolution via Exceptions for Dependable Java Applications, In Proceedings of the International Conference on Dependable Systems and Networks (DSN'03), June, 2003.

[Zuk00] John Zukovski: A Comparative Book Review [on-line] www.javaworld.com. December 2000.