

# Test-First Java Concurrency for the Classroom

Mathias Ricken

Dept. of Computer Science  
Rice University  
Houston, TX 77005, USA  
+1-713-348-3836  
mgricken@rice.edu

Robert Cartwright

Dept. of Computer Science  
Rice University  
Houston, TX 77005, USA  
+1-713-348-6042  
cork@rice.edu

## ABSTRACT

Concurrent programming is becoming more important due to the availability of multi-core processors and the prevalence of graphical user interfaces (GUIs). To adequately prepare students for the concurrent future, instructors have begun to address concurrency earlier in their curricula. Unfortunately, practices like test-driven development that give students a safe footing in single-threaded environments do not extend well into the concurrent domain. This paper describes how ConcJUnit can simplify writing unit tests for multi-threaded programs, and provides examples that can be used to introduce students to concurrent programming.

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent programming.  
D.2.5 [Software Engineering]: Testing and debugging – *Testing tools*.

K.3.2 [Computers and Education]: Computer and information science education – *computer science education*.

## General Terms

Reliability, Languages.

## Keywords

CS education, Java, JUnit, unit testing, concurrent programming, tools, software engineering.

## 1. INTRODUCTION

In test-driven development, tests are written for a unit of code before the code itself is written, and all tests must succeed before a new revision can be committed to the code base, facilitating the early detection and repair of program bugs [7]. This approach has become common in computer science education [6], and adoption in the industry is increasing as well [1][13].

Unfortunately, unit testing is much less effective for programs with multiple threads of control than for single-threaded programs. The importance of concurrent programming, however, increases as CPU clock frequencies stagnate and multi-core

processors become more widespread. To benefit from these new CPUs, programs have to run several computations in parallel. Unless there is a breakthrough in CPU design and manufacturing, writing and testing concurrent code is bound to change from an expert activity to a skill that programmers use every day. Several schools have already responded to this trend and introduce concurrent programming earlier in their curricula [2][3].

Furthermore, multi-threading not only occurs when trying to utilize multi-core CPUs. GUI frameworks, such as Swing, AWT or SWT, access components and react to user input in a separate *event thread*; therefore, GUI applications written using these frameworks already contain aspects of multi-threading.

Developers of larger Java applications, such as DrJava [11], report that the difficulty of applying test-driven development to concurrent programs has two causes: (1) The most frequently used unit testing frameworks make it easy to write bad tests, and (2) thread scheduling is non-deterministic and machine-specific, and the outcome of a test can change from one run to the next [12].

Test-driven design increases programmer confidence [14], which is important especially in introductory programming courses. The fact that tests which should fail may succeed is particularly troubling, since it could give students a false sense of security. It is therefore crucial to discuss the ways in which concurrent unit tests may produce unexpected results and what can be done to address this.

**Contributions** We present a module used to introduce concurrent Java programming at the beginning of a software engineering course often taken right after the CS1/2 sequence. In this module,

1. We observe shortcomings of the popular JUnit [8] framework that especially affect beginners and describe how an extension called ConcJUnit [9] can help alleviate these problems (Section 2).
2. We employ a series of small examples to elucidate problems in concurrent programming. First, we modify a simple counter to perform increments in multiple threads and discuss atomicity when using shared data (Section 3). Our next program requires access to two pieces of shared data and introduces the possibility of deadlock (Section 4). To assess the students' understanding of the covered topics, we assign the implementation of a bounded buffer and a readers-writer lock as homework (Section 5).
3. We discuss why concurrent programs remain difficult to test because of the nondeterministic nature of thread scheduling (Section 6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'10, March 10–13, 2010, Milwaukee, WI, USA.

Copyright 2010 ACM 978-1-60558-885-8/10/03...\$10.00.

## 2. THE NEED FOR AN IMPROVED TESTING FRAMEWORK

The JUnit framework, as well as all other popular unit testing frameworks for Java, behaves in a pathological manner with regard to concurrency: Failed assertions and uncaught exceptions in threads other than the main thread are ignored and do not cause a test to fail. The unit test in Figure 1, for instance, succeeds even though the child thread unconditionally calls JUnit's `fail()` method.

The absence of an exception handler for child threads can easily lead programmers to write tests that succeed even though they should fail. Beginners may be most frequently affected by this, but even experienced programmers may move code out of the main thread: The Swing GUI framework, for example, requires that GUI components and the documents associated with them are accessed in the event thread. The unit testing harness is supposed to provide a rigid foundation for refactoring, but fails to do so if thread boundaries are crossed.

The unit test in Figure 1 contains another design flaw: Even if the failure in the child thread is registered, there is no guarantee that the test has not already ended and been declared a success. JUnit does not warn if some child threads spawned in a test did not actually terminate.

A well-written test ensures that all child threads have ended before the test result is determined. The most common ways to achieve this are (a) using `Thread.join()`, (b) using `Object.wait()` and `Object.notify()`; or (c) by monitoring a shared `volatile` variable and (busy-)looping. While option (c) is usually avoided for performance reasons, both (a) and (b) can be efficiently used to control thread lifetimes. These two options, shown<sup>1</sup> in Figure 2 and Figure 3, are equivalent as long as there are no more instructions after the call to `notify()`.

However, only the first option, using `join()`, ensures that the other thread has truly terminated. With robustness under refactoring in mind, a well-formed unit test should require that all child threads are joined with the test's main thread. This can be done directly, by the main thread invoking `join()` for all child threads, or indirectly, though the transitive property of `join()`, as long as all join operations together ensure that each child thread terminates before the test ends [12].

### 2.1 Unit Testing with ConcJUnit

ConcJUnit is an open-source project and designed as a replacement for JUnit. Since it is backward-compatible, replacing the `junit.jar` file with the appropriate version of ConcJUnit is all that needs to be done to start using the replacement.

ConcJUnit installs a default exception handler for all child threads spawned in the test, as well as for the AWT/Swing event thread. The framework can therefore properly detect uncaught exceptions and failed assertions in any thread. When run with ConcJUnit, the test in Figure 1 fails as expected.

ConcJUnit also enumerates all child threads and causes the test to fail if any of them are still alive. To help the developer determine which threads these are, ConcJUnit records the creation context of

```
1. public void testException() {
2.     Thread t = new Thread() {
3.         public void run() {
4.             // should cause failure but does not
5.             fail();
6.         }
7.     };
8.     t.start();
9. }
```

Figure 1: Test Should Fail But Does Not

```
1. public void testWithJoin() {
2.     Thread t = new Thread() {
3.         public void run() {
4.             // ...
5.         }
6.     };
7.     t.start();
8.     t.join();
9. }
```

Figure 2: Lifetime Control Using Join

```
1. public void testWithJoin() {
2.     final Object sign = new Object();
3.     Thread t = new Thread() {
4.         public void run() {
5.             // ...
6.             synchronized(sign) { sign.notify(); }
7.             // no more code here
8.         }
9.     };
10.    t.start();
11.    synchronized(sign) { sign.wait(); }
12. }
```

Figure 3: Lifetime Control Using Wait/Notify

```
1. public void testException() {
2.     Thread t = new Thread() {
3.         public void run() { /* no op */ }
4.     };
5.     t.start();
6.     Thread.sleep(5000); // long wait
7. }
```

Figure 4: Child Thread Gets Lucky

a thread and pinpoints the exact source location where it was started. For example, in Figure 1 ConcJUnit indicates that the thread that was started in line 8 was still alive.

Furthermore, ConcJUnit analyzes the join operations that were performed and issues a warning when a child thread had terminated before the end of a test, but this was not enforced using a join. For instance, for the test in Figure 4, ConcJUnit reports that the thread started in line 5 ended in time, but this was only due to fortuitous circumstances.

The analysis of the join operations is conservative and follows the notion developed above that all child threads need to be joined with the main thread. As a result, ConcJUnit will emit a warning

<sup>1</sup> `join()` and `wait()` may resume *spuriously* (§17.8.1 JLS [4]). For brevity, the loop necessary to handle this has been omitted.

for a correctly synchronized test, such as the one in Figure 3 employing `wait()` and `notify()`, but these warnings can be suppressed. The conservative analysis is nonetheless useful, since the warnings indicate which tests a programmer should examine by hand.

The thread creation context is helpful for analyzing uncaught exceptions and failed assertions in child threads as well. If a unit test fails because of a problem in a child thread, ConcJUnit will provide the stack trace of the child thread, as well as the stack trace of the parent thread up to the point where the child thread was started (and so on for grandparent threads, etc.). In Figure 5, for example, the failure occurs in a named helper class, and it is not immediately obvious which thread failed, `t1` or `t2`. The extended stack trace with the thread creation context, however, shows that the child thread was started in line 11; therefore, the failure occurred in thread `t2`.

ConcJUnit has recently been integrated into the DrJava IDE and is therefore easy to use. With a testing framework that is better equipped for multi-threaded programs, we now discuss some small examples with our students.

### 3. MULTI-THREADED COUNTER

We explain that concurrency, while helpful for many tasks, becomes most powerful when it is used to break down a computation into smaller pieces, which are processed in parallel by different CPU cores and then combined into a final value. But to produce the end result, there needs to be at least some amount of communication between the threads.

This communication is often performed using shared data. To illustrate this, we reduce the computation to something very basic: A simple counter that increments an integer variable 10 million times serves as placeholder for more complicated computations. At the end of the program, we assert that the counter has indeed reached the expected value. The source code for this unit test can be found in Figure 6. After this test has been written, the students develop the code for the actual counter, shown in Figure 7.

With multiple cores available, it should be possible to distribute the work of incrementing the counter. Figure 8 shows a test that spawns 10 helper threads, each incrementing the counter a million times. The test waits for all helper threads to finish (using the `join()` operation), and then asserts that the integer variable contains the expected number. To the students' surprise, this test fails because the actual value of the counter is less than the expected value. More interestingly, the actual value changes from one execution to the next.

To discover the cause of the failure, we ask the students to describe in detail what operations the expression `count++`; actually performs. They discover that, if one assumes addition can only be performed in a CPU register, the previous line can be re-written in pseudo-code as

```
long register = count;
register = register + 1;
count = register;
```

Shown this way, it becomes evident that the increment expression is not atomic and may be interleaved with operations in other threads. If thread A first reads the counter, but then gets preempted by another thread B also incrementing the counter before A can write the changed value back to memory, then the

```
1. class Helper extends Thread {
2.     boolean toFail;
3.     public Helper(boolean b) { toFail = b; }
4.     public void run() { if (toFail) fail(); }
5. }
6.
7. public void testTCC() {
8.     Thread t1 = new Helper(false);
9.     Thread t2 = new Helper(true);
10.    t1.start();
11.    t2.start();
12.    t1.join(); t2.join();
13. }

junit.framework.AssertionFailedError:
  at TCCTest.Helper.run(TCCTest.java:4)
  at ...parent called Thread.start()...
  at TCCTest.testTCC(TCCTest.java:11)
```

Figure 5: Thread Creation Context for Uncaught Exceptions

```
1. public class SingleIncTest extends TestCase {
2.     static final long N = 10000000;
3.     public void testSingle() {
4.         Counter c = new Counter();
5.         c.incNTimes(N);
6.         assertEquals(N, c.count);
7.     }
8. }
```

Figure 6: Single-threaded Counter Test

```
1. public class Counter {
2.     public long count = 0;
3.     public void incNTimes(long n) {
4.         for(int i=0; i<n; ++i) { count++; }
5.     }
6. }
```

Figure 7: Counter Implementation

```
1. public class MultiIncTest extends TestCase {
2.     static final long N = 1000000;
3.     static final int T = 10;
4.     public void testMulti() throws Exception {
5.         final Counter c = new Counter();
6.         Thread[] ts = new Thread[T];
7.         for(int i=0; i<T; ++i) {
8.             ts[i]=new Thread() {
9.                 public void run() {
10.                    c.incNTimes(N);
11.                }
12.            });
13.            ts[i].start();
14.        }
15.        for(Thread t: ts) t.join();
16.        assertEquals(T*N, c.count);
17.    }
18. }
```

Figure 8: Multi-threaded Counter

```

A1  long register1 = count;    // register1==0
B1  long register2 = count;    // register2==0
A2  register1 = register1 + 1; // register1==1
A3  count = register1;        // count==1
B2  register2 = register2 + 1; // register2==1
B3  count = register2;        // count==1
    final result                // count==1

```

**Figure 9: Problematic Interleaving**

work done by thread A is lost. Figure 9 shows one such problematic interleaving.

This is an example of a *data race*. A data race occurs when (1) two threads access the same shared data, (2) at least one of the accesses is a write access, and (3) there is nothing that prevents the order of the accesses from changing. The accesses in our example are the read access of B1 and the write access of A3: If A3 happens before B1, the program performs as expected; however, if B1 occurs before A3, as shown in Figure 9, then the increment performed by thread A is lost. The actual interleaving is nondeterministic and changes from one run to the next, which also explains why the final value of the counter varies.

To prevent this problem, we need to ensure that the instructions A1; A2; A3 cannot be interleaved with the same instructions in another thread. We do this by introducing a synchronized block requiring a lock object to be acquired before the increment operation may be performed. Threads compete for ownership of the lock object, and only the thread owning it is allowed into the synchronized block protected by the lock. The instructions A1; A2; A3 in the synchronized block are not atomic—thread A may be preempted by other threads—but once thread A has ownership of the lock object, no other thread may execute code protected by the same lock object.

Once the students allocate a lock object as a field of the Counter class and enclose the increment operation in a synchronized block, the unit test will pass.

```

private Object lock = new Object();
// ...
synchronized(lock) { count++; }

```

It is important to emphasize that the code needs to be protected by the same *runtime* object for the synchronized block to be effective; therefore, the lock object cannot be a local variable in the `incNTimes()` method, because a local variable would allocate a new object for every call.

## 4. MULTI-THREADED BANK

After the students have correctly synchronized the multi-threaded counter, we present them with another problem: In a concurrent simulation of a bank's checking accounts, we would like to make random transfers from one account to another. This implies subtracting a value  $x$  from the balance of account  $A$  and adding it to the balance of account  $B$ . We use the notation  $(A \rightarrow B, x)$  for this transfer.

One drastic approach would be to have one lock object protecting the access to all accounts; however, this would eliminate concurrent transfers and essentially serialize the program. We would like to allow several transfers at the same time, as long as they don't involve the same accounts. For instance, it should be allowed to execute the transfers  $(0 \rightarrow 1, 10)$  and  $(2 \rightarrow 3, 20)$

```

1.  final int NUM_ACCOUNTS = 5;
2.  final int NUM_TRANSFERS = 1000000;
3.  long[]   accounts = new long[NUM_ACCOUNTS];
4.  Object[] locks = new Object[NUM_ACCOUNTS];
5.  Random   r = new Random();
6.  class TransferThread extends Thread {
7.      public int from, to;
8.      public void run() {
9.          for(int i=0; i<NUM_TRANSFERS; ++i) {
10.             from = r.nextInt(NUM_ACCOUNTS);
11.             to = r.nextInt(NUM_ACCOUNTS);
12.             synchronized(locks[to]) {
13.                 synchronized(locks[from]) {
14.                     int x = r.nextInt(100);
15.                     accounts[from] -= x;
16.                     accounts[to] += x;
17.                 }
18.             }
19.         }
20.     }
21. }

```

**Figure 10: Nested Synchronized Blocks**

```

1.  class CheckThread extends Thread {
2.      public void run() {
3.          while(true) {
4.              Thread.sleep(5000);
5.              for(TransferThread t: ts) {
6.                  System.out.println(t.from+"->"+t.to);
7.              }
8.          }
9.      }
10. }

```

```

Output:  0->1
         1->0

```

**Figure 11: Diagnostic Background Thread and Output**

concurrently, but the transfer  $(1 \rightarrow 2, 15)$  should block until accounts 1 and 2 are not in use anymore.

Most students will suggest one lock object per account acquired in two nested synchronized blocks, as shown in Figure 10. It is worthwhile to point out that Java locks are *re-entrant*, i.e. if a thread has already acquired ownership of a lock, attempting to do so again is a no-op; therefore, it is not problematic if `from == to`. When the students run a unit test that issues random transfers and checks that the total amount of money remains constant, it is almost certain that the program will “hang” after a few seconds without finishing all transfers.

Unfortunately, it is difficult to determine what exactly happened without using additional tools such as a debugger. Adding a print statement after line 8 to show the values of `from` and `to` will not be insightful because performing console output inadvertently synchronizes the program and avoids the problem. However, it is possible to run an additional diagnostic thread, shown in Figure 11, in the background and periodically print the current `to` and `from` values of all the transfer threads. When the students analyze the diagnostic output, they realize there is a cycle: For instance, one thread has claimed lock object 0 and needs lock object 1,

while the other thread owns lock object 1 and requires lock object 0; neither thread can proceed. This situation is called *deadlock*.

It is interesting to note that there would be no deadlock if the second thread had also attempted to procure lock object 0 first. In that case, the second thread would just have to wait until the lock object is released again. The order in which the lock objects are sought is clearly important. This leads students to the realization that lock objects always need to be acquired in the same order.

In our account transfer example, one way to ensure this is to always acquire the lock with the lower account number first:

```
synchronized(locks[Math.min(to, from)]) {  
    synchronized(locks[Math.max(to, from)]) {
```

The actual order is arbitrary, of course, as long as one exists.

## 5. HOMEWORK ASSIGNMENT

To assess the students' understanding of the topics covered, we assign the implementation of a bounded buffer and a readers-writer lock as homework. We provide unit test suites for both, allowing students to test their programs before they turn in their assignments.

While we mostly grade for correctness, which among other things involves checking that the implemented methods are correctly synchronized, we also consider efficiency and fairness. For example, only as few threads as possible should be woken up when the bounded buffer becomes non-full or non-empty. For the readers-writer lock, readers and writers should be put in a queue to ensure that readers cannot starve writers.

The assignment, along with the solutions, is available at [10]. Note that we also discuss synchronized methods, volatile variables, and the `Object.wait()`, `Object.notify()` and `Thread.interrupt()` methods before we assign this homework. Brief notes are contained in the assignment.

## 6. NONDETERMINISTIC SCHEDULING

We also point out to our students that ConcJUnit can only detect problems in the schedule (the way threads were interleaved) that was chosen [12]. It cannot detect all defects that *could* occur; only those that happened in the schedule that was actually executed are found. Even if the test suite passes ConcJUnit without failures or warnings, it is still possible that the unit tests fail during the next run. This is due to the nondeterministic nature of thread scheduling.

Unit testing requires that known input generates known output, and that the observed results of the computation are deterministic, a property that is lost for concurrent programs. A data race, for example, introduces non-determinism. We should therefore require that all accesses to the same shared variable are consistently protected by the same lock object. While some data races may be benign, many of them are bugs; it is therefore reasonable to treat a data race as a defect unless the developer instructs the test framework to ignore the data race. There are automatic data race detectors available for Java, like ConTest [5], for instance.

Unfortunately, even a data race-free program can still behave non-deterministically. Thread scheduling depends on many factors that are out of the developer's control, such as system load and the state of I/O devices. A test that executes two computations in parallel and then transfers the results into a synchronized queue

may end up with two different results,  $(a, b)$  and  $(b, a)$ , even if there was no race accessing the queue, the only shared variable.

It would be useful to ensure that a unit test passes under all possible schedules, but the number of schedules increases exponentially with the size of the program. A heuristic could choose a representative subset of schedules to execute, or the test could be executed many times while the schedule is modified by inserting delays at critical synchronization points of the program. This is still the subject of ongoing research.

## 7. CONCLUSION

With multi-core CPUs replacing older single-core designs, concurrent programming will become more prominent. It is therefore important to educate students about the difficulties that arise in multi-threaded programs. Using ConcJUnit and the examples presented in this paper, instructors can moderate the conflict between proven test-driven development and emerging concurrent programming.

## 8. REFERENCES

- [1] Bowers, A. N., Sangwan, R. S., and Neill, C. J. 2007. *Adoption of XP practices in the industry—A survey: Research Sections*. *Softw. Process* 12, 3 (May. 2007), 283-294.
- [2] Ernst, D. J. and Stevenson, D. E. 2008. *Concurrent CS: preparing students for a multicore world*. In Proceedings of the 13th Annual Conference on innovation and Technology in Computer Science Education (Madrid, Spain, June 30 - July 02, 2008). ITiCSE '08. ACM, New York, NY, 230-234.
- [3] Fekete, A. D. 2009. *Teaching about threading: where and what?*. *SIGACT News* 40, 1 (Feb. 2009), 51-57.
- [4] Gosling, J., Joy, B., Steele, G. L. and Bracha, G. *The Java Language Specification*. 3rd Ed. Addison-Wesley, Reading, MA, USA, 2005.
- [5] IBM. *ConTest*. <http://www.alphaworks.ibm.com/tech/contest>
- [6] Jeffries, R. and Melnik, G. 2007. *Guest Editors' Introduction: TDD—The Art of Fearless Programming*. *IEEE Softw.* 24, 3 (May. 2007), 24-30.
- [7] Jeffries, R. <http://www.xprogramming.com/>
- [8] JUnit Project. *JUnit*. <http://junit.org/>
- [9] Rice JavaPLT. *ConcJUnit*. <http://concutest.org/concjunit/>
- [10] Rice JavaPLT. *Concurrent programming assignment and solution*. <http://concutest.org/download/sigcse2010-java-concurrency.zip>
- [11] Rice JavaPLT. *DrJava Web Site*, <http://drjava.org/>
- [12] Ricken, M. and Cartwright, R. 2009. *ConcJUnit: unit testing for concurrent programs*. In Proceedings of the 7th international Conference on Principles and Practice of Programming in Java (Calgary, Alberta, Canada, August 27 - 28, 2009). PPPJ '09. ACM, New York, NY, 129-132.
- [13] Rumpe, B. and Schroeder, A. *Quantitative Survey on Extreme Programming Projects*. In Proceedings of International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002). (Alghero, Italy, May 2002) 95-100.
- [14] Spacco, J. and Pugh, W. 2006. *Helping students appreciate test-driven development (TDD)*. In Companion To the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 907-913.