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 growing dominance of multi-core processors and the prevalence of graphical user interfaces (GUIs). To prepare students for the concurrent future, instructors have begun to address concurrency earlier in their curricula. Unfortunately, test-driven development, which enables students and practitioners to quickly develop reliable single-threaded programs, is not as effective in the domain of concurrent programming. This paper describes how ConcJUnit can simplify the task of 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 to software development is steadily gaining popularity both in computer science education [6] and industrial practice [1][13].

Unfortunately, unit testing is much less effective for programs with multiple threads of control than for sequential (single-threaded) programs. The importance of concurrent programming, however, is rapidly growing as multi-core processors replace older single core designs. To exploit the power of these new processors, programs must run several computations in parallel. Unless there is a breakthrough in processor design or language implementation technology, writing and testing concurrent code will become a skill that all software developers must master. Several schools have already responded to this trend and introduced concurrent programming concepts early in their curricula [2][3].

Furthermore, multi-threading not only occurs in applications designed to exploit multi-core CPUs. GUI frameworks like AWT/Swing and SWT access components and react to user input in a separate *event thread*. As a result, most applications with GUIs already involve multi-threading.

Developers of large Java applications like DrJava [11] have identified two obstacles to applying test-driven development to concurrent programs: (*i*) the standard unit testing frameworks make it easy to write bad tests and (*ii*) thread scheduling is non-deterministic and machine-specific, implying that the outcome of a test can change from one run to the next [12].

Test-driven design increases programmer confidence [14], which is especially important in introductory programming courses. The fact that tests with failed assertions may succeed is particularly troubling, because it could give students a false sense of security. It is therefore crucial to identify how concurrent unit tests may report false successes and what can be done to address this issue.

Contributions In this paper, we present a course module introducing concurrent Java programming, which is suitable for inclusion at the beginning of a software engineering course that only presumes the CS 1/2 sequence as prerequisite. In this module:

1. We identify the shortcomings of the standard JUnit [8] framework (and its competitors [15]) in the context of concurrency and describe how an extension of JUnit called ConcJUnit [9] remedies these problems (Section 2).
2. We present a series of small examples that elucidate common problems in concurrent programming. First, we study a trivial program that increments a counter from multiple threads to stress the importance of atomicity when performing operations on shared data (Section 3). Next, we write and analyze a program that requires access to two pieces of shared data, introducing 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 nondeterministic thread scheduling (Section 6).

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

2. IMPROVING TESTING FRAMEWORKS

All popular unit testing frameworks for Java including JUnit behave pathologically 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.

Inexperienced programmers unfamiliar with this pathology typically write unit tests for multi-threaded units of code that report success when assertions in auxiliary threads fail. Even experienced programmers may fall prey to this problem when they move code out of the main thread. The AWT/Swing and SWT GUI frameworks, for example, mandate that GUI components and the associated documents be accessed from the event thread. A 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 reveals another design flaw in JUnit: even when a test creates an exception handler for a child thread, there is no guarantee when the child thread fails that the test has not already ended and falsely reported as a success. JUnit does not warn if some child threads spawned in a test do not terminate on time.

A well-written test ensures that all child threads have ended before the test outcome is determined. The most common ways to achieve this property are (a) using `Thread.join()`, (b) using `Object.wait()` and `Object.notify()`; or (c) by monitoring a shared `volatile` variable in a busy-loop. 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[1] in Figure 2 and Figure 3, are equivalent as long as the child thread terminates immediately after the call to `notify()`.

However, only the first option, using `join()`, ensures that the other thread has truly terminated. To support robustness under refactoring, a well-formed unit test should require that all child threads are joined with the test's main thread. This action can be done directly, with the main thread invoking `join()` for all child threads, or indirectly, through 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 designed as a replacement for JUnit. Since it is backward-compatible with JUnit, replacing the `junit.jar` file with the appropriate version of ConcJUnit enables support for concurrent unit testing while preserving existing testing behavior.

ConcJUnit installs a default exception handler for all child threads spawned in a 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 tracks all child threads and causes the test to fail if any of them are still alive when the main thread terminates. To help the developer determine which threads these are, ConcJUnit records the source location where each child thread is started and reports this information for any thread that does not properly terminate. For example, in Figure 1, ConcJUnit reports that the thread started in line 8 did not terminate.

Furthermore, ConcJUnit analyzes the join operations that are performed and issues a warning when a child thread terminates but is not absorbed by a join. For instance, for the test in Figure 4, ConcJUnit reports that the thread started in line 5 terminated, but only because of the vagaries of thread scheduling.

The analysis of join operations is conservative and enforces the policy that all child threads must be joined with the main thread. As a result, ConcJUnit will emit a warning for some correctly synchronized tests, such as the one in Figure 3 employing `wait()` and `notify()`. At the programmer's option, these warnings can

```
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/Nofity

```
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

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

be suppressed. In our experience, the conservative analysis is helpful, because it catches improperly synchronized tests. Moreover, it is easy to ensure that child threads terminate using join operations.

Thread creation coordinates are helpful in analyzing uncaught exceptions and failed assertions in child threads as well. If a unit test fails in a child thread, ConcJUnit provides a stack trace of the failed child thread, as well as the stack trace of all ancestor threads up to the point where the child thread was started. 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, making it easy for beginners to use. Given a testing framework that is well equipped for multi-threaded programs, we subsequently discuss some small examples with our students.

3. MULTI-THREADED COUNTER

We explain that concurrency, while helpful for many tasks, improves performance 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. Communication between threads is required to coordinate the assembly of the final value from the results produced by the individual threads.

This communication is usually performed using shared data. To illustrate this technique, we present a very simple example: a computation that increments an integer counter 10 million times. We initially write a single-threaded test program for this task. At the end of the program, we assert that the counter has indeed reached the expected value. The source code for this unit test is given in Figure 6. After this test has been written, the students develop the code for the actual counter, shown in Figure 7.

When multiple cores are available, it should be possible to distribute the work of incrementing the counter. Figure 8 shows a test program 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 typically 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 work done by thread A is lost. Figure 9 shows one such problematic interleaving.

```
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

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) the accesses are unsynchronized (nothing 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 eliminate this concurrency bug, 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 the value of a local variable in the `incNTimes()` method, because a local variable will be bound to a different object in each 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 arbitrary 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→1, 10) and (2→3, 20) concurrently, but the transfer (1→2, 15) should block until accounts 1 and 2 are not in use anymore.

```
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

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 a student runs 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 11 to show the values of `from` and `to` is not helpful because performing console output inadvertently synchronizes competing threads, eliminating the "hanging" problem in typical schedules. 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 critical. This observation leads students to the realization that lock objects must be acquired according to the same global 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]) {
```

Any total order on lock objects will work.

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 primarily grade for correctness, which among other things involves checking that the implemented methods are correctly synchronized, we also consider efficiency and fairness. For example, a bounded buffer implementation should maximize concurrency, yet avoid awakening more waiting producers or consumers than can be accommodated. 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` and `final` 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 inform our students that ConcJUnit can only detect problems in the schedule (the interleaving of operations on shared data) [12] chosen by the JVM for each test. ConcJUnit does not detect all defects that *could* occur; only those in the executed schedule are found. Even when a ConcJUnit test passes without failures or warnings, the same test could fail on the next run.

Conventional unit testing assumes the program behavior is deterministic, a property that is lost for concurrent programs. The ordering of competing accesses to shared data is non-deterministic – even when those accesses are synchronized. The Java Memory Model [5] does not even ensure that program execution corresponds to a serial interleaving of its threads unless the program is free of data races. The simplest strategy for avoiding data races in Java is to mark all shared variables as `final` or `volatile`. Both dynamic and static data race detectors have been developed for Java [5], but this is still an active area of research. We are not aware of any lightweight, practical race detection tools suitable for use in the classroom or in routine software development.

Since concurrent program execution is non-deterministic, a unit testing framework should ideally run each test under all possible schedules. Unfortunately, the number of possible schedules increases exponentially with the size of the program. A practical alternative to exhaustively running each test under all schedules might be to run each test under a set of schedules generated using heuristic methods. The design and construction of such a heuristic tool is one of our primary research interests.

7. CONCLUSION

As mainstream processor designs become more explicitly parallel and include more cores, concurrent programming will become more prominent in both computer science education and industrial practice. It is therefore essential to educate students about the challenges involved in writing efficient and reliable concurrent programs. Using ConcJUnit and the examples presented in this paper, instructors can temper the conflict between test-driven development and 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

[15] TestNG Project. *TestNG*. http://testng.org/