# The Craftsman: 10
# SocketService 5
# Dangling Threads

Robert C. Martin
14 Jan, 2003

*...Continued from last month. See <<link>> for last month's article, and the code we were working on. You can download that code from:*

*www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR4_TestMultiThreaded.zip*

---

I have a standing monthly reservation for breakfast on the observation deck. It's a bit of an extravagance on an apprentice's credit, but I love eating under the starbow.

As I ate, I started thinking about the dangling thread problem we solved yesterday. We fixed the `serverThread`, but the serviceRunnable threads were all still left dangling. I was certain Jerry would want to fix those before moving on to other issues.

Sure enough, when I walked into the lab, Jerry was already there with the following test case on the screen:

```
  public void testAllServersClosed() throws Exception {
    ss.serve(999, new WaitThenClose());
    Socket s1 = new Socket("localhost", 999);
    Thread.sleep(20);
    assertEquals(1,WaitThenClose.threadsActive);
    ss.close();
    assertEquals(0, WaitThenClose.threadsActive);
  }
```

"Ah, I see what you are doing." I said. "You are making sure that all `SocketServers` are closed by the time you return from closing the `SocketService`."

"Right! I want to make sure we don't leave those servers dangling."

"But you are only testing it with one server. Don't we need to test it with many?"

"Right again! Lets get this simple test working first and then we'll add more servers to it."

"OK", I said, "I think I know how to write `WaitThenClose`."

```
class WaitThenClose implements SocketServer {
  public static int threadsActive = 0;
  public void serve(Socket s) {
    threadsActive++;
    delay();
    threadsActive--;
  }

  private void delay() {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
    }
  }
}
```

As I wrote this code, Jerry nodded in concurrence as it appeared on the screen. This was just what he had thought `WaitThenClose` would be like. Once it was done and compiled we ran the tests, and they failed as expected:

```
1)  testAllServersClosed AssertionFailedError: expected:<0> but was:<1>
```

Jerry rubbed his hands together and said: "OK, now lets make it pass the test." He made a move to grab the keyboard but I blocked it saying: "I think I have an idea about that." So while Jerry watched, I made the following changes.

```
private LinkedList serverThreads = new LinkedList();

public void serve(int port, SocketServer server) throws Exception {
  itsServer = server;
  serverSocket = new ServerSocket(port);
  serverThread = new Thread(
    new Runnable() {
      public void run() {
        running = true;
        while (running) {
          try {
            Socket s = serverSocket.accept();
            Thread serverThread = new Thread(new ServiceRunnable(s));
            serverThreads.add(serverThread);
            serverThread.start();
          } catch (IOException e) {
          }
        }
      }
    }
  );
  serverThread.start();
}

public void close() throws Exception {
  if (running) {
    running = false;
    serverSocket.close();
    serverThread.join();
    for (Iterator i = serverThreads.iterator(); i.hasNext();) {
      Thread thread = (Thread) i.next();
      serverThreads.remove(thread);
      thread.join();
    }
  } else {
    serverSocket.close();
  }
}
```

As I compiled this code, Jerry got a funny look on his face.

"Is this OK?" I asked.

"Let's see." He said. "Run the test."

When I ran test it gave me a peculiar error:

```
1) testOneConnection java.util.ConcurrentModificationException
```

"What's that?" I said.

"You've broken a rule, Alphonse. You never add or remove something from a list while you have an iterator active."

"Ouch! Of course! I knew that!" I said sheepishly. "OK, but that's easy to fix because I really don't need to remove the thread from the list, do I?" I quickly removed the `remove` line, and ran the test again.

"Ah! Now it's working just fine."

Jerry nodded, but stared at me expectantly.

"What!?" I exclaimed after enduring his gaze for half a minute.

"You are still modifying the list while the iterator is active."

"I am?" I was quite puzzled by this. "Jerry, there's only one place where the list is modified, and that's where the thread is added in the `running` loop. How can that be called while the iterator is active?"

"Oh, I think it's possible. The call to accept might be on the verge of returning as you enter the iterator loop. When the iterator loop blocks on a join, the accept will return and add a thread to the list."

"OK, I can sort of see that, but can we test for it?"

"We could, but there's no point. It turns out that there's another place where you'll be modifying the list while the iterator is open."

"There is?"

"Yes, you are just about to add it. How many threads are in that list?"

"All of them...Oh! I should be deleting the thread from the list when the thread completes! Otherwise completed threads will just hang around in the list." I grabbed the keyboard and made the following change.

```
class ServiceRunnable implements Runnable {
  private Socket itsSocket;

  ServiceRunnable(Socket s) {
    itsSocket = s;
  }

  public void run() {
    try {
      itsServer.serve(itsSocket);
      serverThreads.remove(Thread.currentThread());
      itsSocket.close();
    } catch (IOException e) {
    }
  }
}
```

"Aha! Now it fails again." I said. "You were right! Some of the threads complete before the iterator loop finishes. Boy, it's a good thing the iterator complains about concurrent updates!"

"Yes, it is." Nodded Jerry. "Now let me show you how I'd fix this."

Once again he took the keyboard, and once again I did not protest. He proceeded to make the following change.

```
public void close() throws Exception {
  if (running) {
    running = false;
    serverSocket.close();
    serverThread.join();
    while (serverThreads.size() > 0) {
      Thread t = (Thread)serverThreads.get(0);
      serverThreads.remove(t);
```

```
        t.join();
      }
    } else {
      serverSocket.close();
    }
  }
```

"There!" Jerry said. "Now all the tests pass again."

"I get it!" I said. "Instead of using an iterator, you just pull the first element off the list and keep looping until the list is empty."

"Right. That way there's no iterator open for a long period of time. Joins can take awhile. It's not good to leave iterator open for long periods when other threads might try to modify the list."

"So are we done?"

Jerry shook his head. "No, there's still a danger."

"What do you mean? What's wrong now?"

"Alponse, whenever you have a container being modified by many different threads, there is a chance that the two threads will collide inside the container. One thread might be in the middle of adding an element while another thread is in the middle of deleting one. When that happens the container can get corrupted and bizarre things can start happening."

"Oh, so you mean we should synchronize access to the container?"

"Yes, that's exactly what I mean. We need to make sure that while the container is being modified, no other thread can access it."

"OK, I can do that. It's pretty simple!" So I proceeded to surround the `add` statement and the two `remove` statements with `synchronized(serverThreads){...}`. I ran the tests, and they all still worked.

"That's one way." Jerry said with a smile. "But it's a bit error prone. If someone else modifies the code and puts in another add or remove then they have to remember to put the `synchronized` statements in place. If they forget then bad things could happen."

I thought about that for a minute and decided that he was right. It would be better if we didn't have to surround every statement that manipulated the list with synchronization. "You know a better way then?"

"Yes." He took the keyboard, took out all my `synchronized` statements. Then he changed one more line of code -- the line that created the `LinkedList` in the first place:

```
  private List serverThreads = Collections.synchronizedList(new LinkedList());
```

Jerry compiled the code and ran all the tests. Everything worked just fine. Then he looked at me and asked: "What do you know about design patterns, Alphonse? Have you heard of the Decorator pattern?"

"I've heard of them, of course, and I've seen the book on people's bookshelves, but I don't know too much about them."

Jerry looked sternly at me and said: "Then it's time you started learning about them in earnest. You can borrow my book and study it if you like. The first think I want you to read is the chapter on the Decorator pattern. The `synchronizedList` function that we just called wraps the `LinkedList` in a Decorator. All calls to the LinkedList are synchronized by it."

"That sounds like a pretty good solution." I said.

"Yeah, well, you just have to remember to explicitly synchronize any use of an iterator." He grimaced.

"Really? You mean iteration isn't synchronized in a synchronized list?"

"TANSTAAFL!" He replied?

"Huh?"

"TANSTAAFL! There Ain't No Such Thing As A Free Lunch."

"I know." I said as I headed off toward the galley for lunch.

*The code that Jerry and Alphonse finished can be retrieved from:*

*www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR4_TestMultiThreaded.zip*