- While handling an IO event, the `SelectionKey` will still signal that it's ready. This can result in multiple threads calling into the same handler if you don't clear that operation from the key's interest set.
- The leader must perform all changes to a `SelectionKey`'s interest set or else you get race conditions with the `Selector`, so we had to build a queue of pending `SelectionKey` changes that the leader thread would execute before calling select.

Handling these tricky details led to quite a bit more coupling between the various objects than I initially expected. If we had been building a framework, this whole area would have needed much more attention to loose coupling. For an application, however, we felt it was acceptable to regard the collection of collaborating objects in the server as a cohesive unit.

One particularly interesting effect showed up only when we ran a packet sniffer to see if we were really getting the maximum possible throughput. We weren't. At first, when the reactor read from a socket that had data available, it would read one buffer full and then return. We figured that it wouldn't take very long to get back around the loop if more than 8,192 bytes were available. It turns out that the studio network is fast enough to fill the server's TCP window before the next thread could get back into the handler, so virtually every transfer would stall for about half of the total transfer time. We added a loop inside the reactor, so it would keep reading until the buffer was drained. That cut the transfer time by nearly half, and reduced the amount of overhead in threading and dispatching. I found this particularly interesting because it works only for fast networks with low latency and only if the total number of clients is small. With higher network latency or more clients, looping that way would risk starving some clients. Again, it was a trade-off that made sense in our context.

UNIT TESTING AND CODE REVIEW

This NIO file server was the one time that I found it helpful to do a large group review, even on an agile project with complete pairing.

My pair and I worked on the threading, locking, and NIO mechanisms over most of an iteration. We unit tested what we could, but between the threading and low-level socket IO, we found it difficult to gain confidence in the code. So we did the next best thing: we got more eyes on it. I'd call that a special case, though. We were compensating for our inability to write sufficient unit tests.

In general, having two sets of eyes on the code all the time provides all the benefits of a code review. Combine that with automatic formatting and style checking, and there's just not enough remaining advantage of a code review to offset its cost. And if you can get the benefits without the cost, then why bother with the code review?

We kept a projector in our lab, connected to two machines through an A/B switch. Whenever we had a technique to illustrate or a design pattern to share, we'd take a few minutes after lunch to fire up the projector and walk through some code. This was particularly handy during the early stages, when