Sample test construction:
In testing the performance and reliability of each model, I used a sample test that divided up the work into 20 threads (chosen arbitrarily), performed 10000000 swaps (I found that in testing values even larger than 10000000, *GetNSet* would often fail to terminate. I believe this to be related to a change in the array values that would prevent a true return value in swap, causing the program to fail to terminate), a max range of 100, and 10 random values for the array. I created these testing large testing parameters to allow for greater variations in my swaps and a more pronounced multithreading result (as more threads access the same elements), resulting in more distinct performance/reliability results across each model. The performance measure for each model is measured as an average over all successful runs out of 5 total runs; the reliability measure is measured as an average total sums from all 5 runs (out of original value 450).

Performance/reliability measurements:

MODEL	TIME(NS/TRANSITION)	SUM MISMATCH
Null	194.4406 (0 fails)	N/A
Synchronized	3995.71 (0 fails)	N/A
Unsynchronized	859.064 (2 fails)	867.74333
GetNSet	3129.525 (3 fails)	922
BetterSafe	1726.08 (0 fails)	N/A
BetterSorry	273.089 (0 fails)	N/A

SynchronizedState: *SynchronizedState* is DRF as the *Synchronized* keyword prevents thread interference and memory consistency errors from multiple threads reading and writing to the same location in memory.

UnsynchronizedState: *UnsynchronizedState* is not DRF as illustrated by the sum mismatch errors that were produced by the sample test parameters above; this is because, without the keyword *Synchronized,* this allows for multiple threads to access the elements of *UnsynchronizedState*, such as in the *swap* method which reads and modifies the elements *value[i]* and *value[j].* The implementation for *UnsynchronizedState* was relatively easy.

GetNSet: *GetNSet* is not DRF as illustrated by the sum mismatch errors that were produced by the sample test parameters above; this is in due part to the *get* methods within the *swap* function. Although these *get* functions are themselves atomic, their sequential orderings/execution is not threadsafe. In implementing *GetNSet*, I first had trouble with implementing the *current* method. After finding out

Richard Chung
804050445

that I could not simply return the AtomicIntegerArray *value* by simple type-casting it as (byte), I experimented with several methods before type casting each element first and then putting them into an instantiated return list. Then, I used the *set* functions native to the *AtomicIntegerArray* library to increment and decrement the *value* variables.

BetterSafe: *BetterSafe* is DRF as illustrated by: (1) the non-existence of sum mismatch errors and (2) the use of lock mechanisms in my code. In implementing a *private final* lock variable that is locked and unlocking in *swap*, the function that made *GetNSet* not DRF, each *value* element is accessed/modified by one thread at a time. This preserves 100% reliability in running the code and performs better than Synchronized, although in documentation, both prevent multiple concurrent accesses to the same area in memory by multiple threads. I had initially though that this lock implementation would make *BetterSafe* slower than *SynchronizedState* due to the additional overhead; however, with the dramatic performance improvements, I think that it is because *Reentrantlock* objects do not have the block locking structure that *Synchronized* does, but rather, an unstructured locking approach that is more appropriate for the increment/decrement/read/write operations of the program.

BetterSorry: In my implementation of BetterSorry, I was not able to provided an implementation that was faster than BetterSafe and, at the same time, not DRF. At first, I had tried an implementation that took advantage of the atomic nature of *AtmoicIntegerArray* (inspired by *GetNSet,* using atomic increment/decrement functions) based on the intuition that atomic objects had less overhead that both locks and synchronization (from documentation). However, this approach ended up being less fast than BetterSafe. My implementation does, however, perform better than BetterSafe and GetNSet while maintaining a high level of reliability (possibly 100%). In *BetterSorry,* I created a *private ReentrantLock* array, capitalizing upon its *isHeldByCurrentThread* function in my *swap* function to prevent concurrent race conditions; values are only modified if they are held by the same thread. Although I could not find any conclusive tests to prove that my implementation is not DRF, one race condition that could possibly exist would be concurrent accesses to the *value* objects in the *swap* method (as explained above).

Given my data, I would use *BetterSorry* for GDI applications (due to its high performance/reliability ratio). However, given that there may be a data race, I would use *BetterSafe*. However, one must note that these values are subjective to the running system; the results are relative to each other and not concrete as I got different values when running my program at different times.