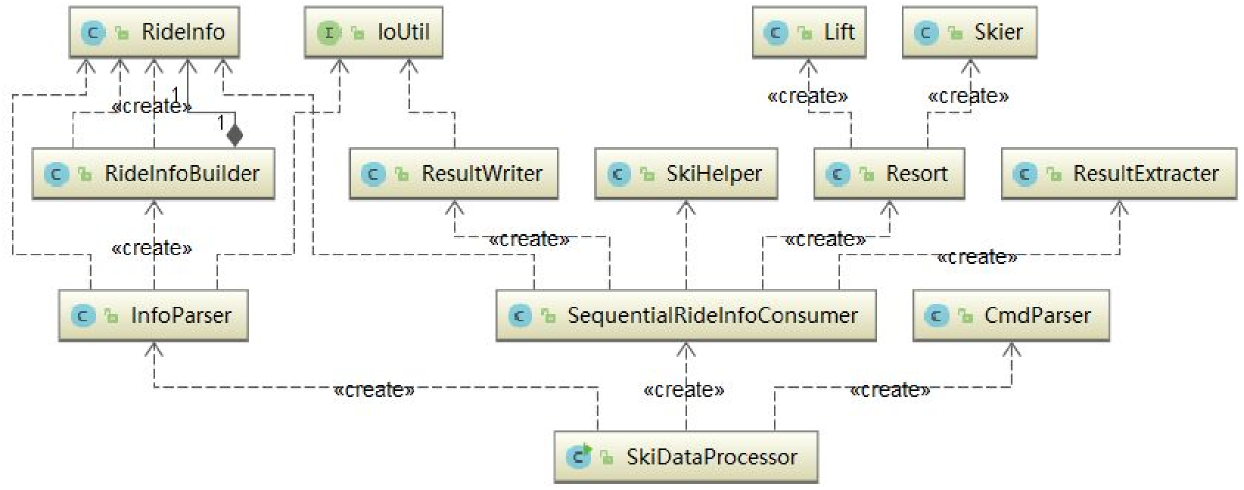
For the sequential solution, the **SkiDataProcessor** will generate **SequentialRideInfoConsumer** object, then pass it to **InfoParser**. InfoParser then parses the information line by line to **RideInfo** using **RideInfoBuilder** and convert RideInfo to update the state of **Lift**s and **Skier**s in **Resort**. After parsing the whole input csv file, all the information is stored in Resort. We can use **ResultExtracter** to extract three questions' result seperately. Finally, use **ResultWriter** to write results to output csv files.

How to parse information:
For Skier, it has a field totalVertical. For each piece of ride information, we add vertical value corresponding to the lift id to totalVertical of the Skier.
For Lift, it has fields of totalRides and ridesForHour. ridesForHour is a list of Integer, it's index specify the ride is in which hour, and the value states the number of rides in that hour. For each piece of ride information, we increse the value of totalRides for specific lift by 1, and also increase the corresponding value in ridesForHour.
After parsing all rows of input csv file, all the information will be stored in Skiers and Lifts. Thus, we can simply use Resort to calculate the results.

UML for Sequential

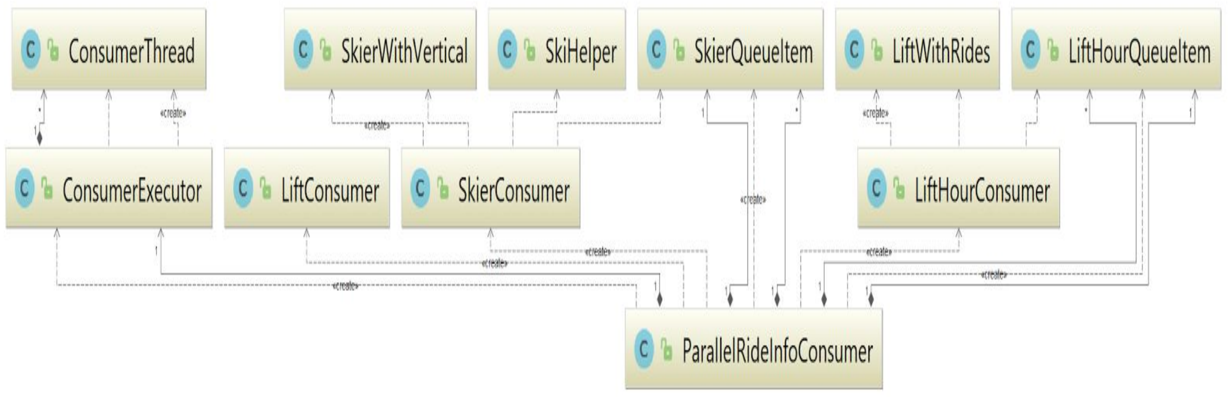


Concurrent:
Depending on flag-concurrent/sequential, we make an appropriate IRideInfoConsumer (ParallelRideInfoConsumer for concurrent and SequentialRideInfoConsumer for sequential), And then we parse information by calling InfoParser and we use builder pattern to parse the row extracted out of file as there are so many arguments and thus builder pattern is better than taking constructor of this many arguments. And when ParallelRideInfoConsumer accepts the row, it gets skier, hour, lift information out of row and then adds those items to appropriate queues (adds skier and lift info. To skierQueue, adds lift info to liftQueue, adds lift and time info to liftHourQueue). All the three queues taken are ConcurrentLinkedQueues.

So, after parsing of information is done, we have all the appropriate information in three queues. And after parsing is complete, we start the three consumer execution by calling - skierConsumer, liftConsumer, liftHourConsumer. These three are consumer executors. Consumer processes will take three valid arguments - queue to act on, Consumer object, and number of threads. And we have used generics so that we dont have to make three separate consumer executors for executing three queues. And writing to files is also done concurrently. SkierConsumer, LiftConsumer, LiftHourConsumer have an accept method that gets appropriate queues-and start writing results to a list of string array, where every entry is one result. And after processing is complete(i.e all results generated), we call fileWriter class to write results to file and along with result, we also pass it a csv file name to write results to.

Optimum Number of threads :
Providing 1 thread to each of three queues of skier/vertical, lift and lift/hour so that these three processes each occur using one thread each. Increasing the consumer thread for processes increases the time consumption because our process is I/O Bound, And we think further reason for this is context -switching. If a number of threads are given the same priority they need to be switched around till they finish execution. And In our case, when there are more threads a lot of context switching takes place when compared to just running less threads.
1 Thread :
time taken to read rows: 845 ms
Time taken for Concurrent : 1198 ms
2 Threads :
time taken to read rows: 1466 ms
Time taken for Concurrent : 1945 ms
3 Threads :
time taken to read rows: 2316 ms

Time taken for Concurrent : 2786 ms


ConsumerThread
SkierWithVertical
SkiHelper
SkierQueueItem
LiftWithRides
LiftHourQueueItem
ConsumerExecutor
LiftConsumer
SkierConsumer
LiftHourConsumer
ParallelRideInfoConsumer
«create»