Jonathan Tabac II c3267035
Seng2200 Assignment 3 Report

Time Spent on Project

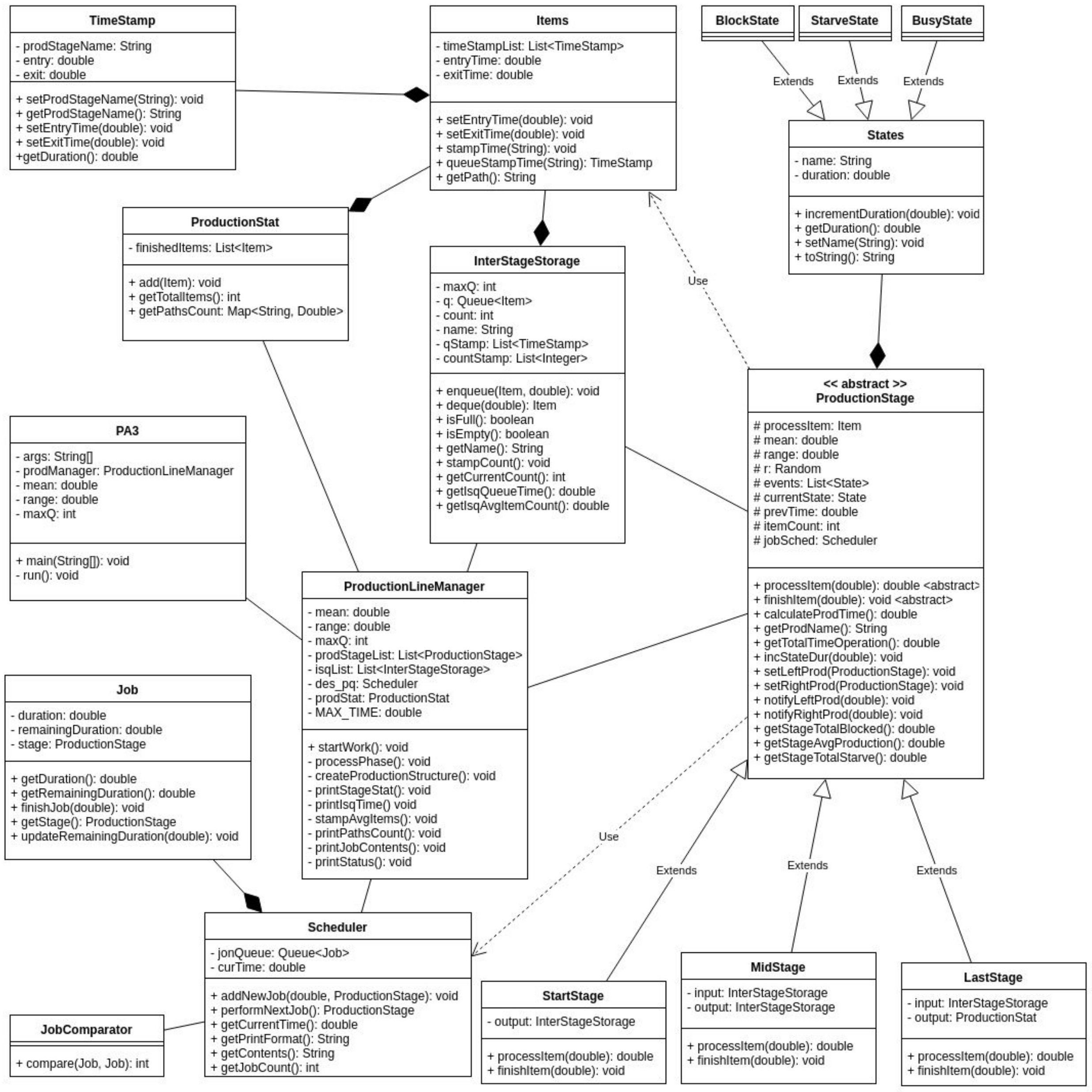
Design - Around 8 hours were spent on designing, which include understanding the specs and drawing up a logical solution with appropriate classes that somehow have separate concerns to promote cohesion within classes and less coupling among them. Discussion among peers were also done all throughout the development of this assignment.

Coding - The initial code was done after the design, which totals to around 5 hours. Coding did not include rework of the code due to bugs and design flaws. It was a good exercise to practice implementing logical and useable inheritance that make sense.

Testing - The most difficult part of the assignment was testing. There were no fixed correct output that was provided (which will most likely happen in the real world) so it was difficult to confirm if the output of the simulation were correct. Collaboration among peers helped solve this by seeing if my results is the odd one out. This helped confirm logical flaws and in understanding deeper the Discrete Events Simulation in general. A good 16 hours were spent on testing.

Correcting - The most satisfying part of the assignment is correcting because after spotting logical bugs in the implementation after long tedious testing and stepping through the code, correcting it is just a matter of a small adjustment in the design (if needed, which I did) and coding it up. After which, I can be assured that my code has one less bug in it than before. A good 10 hours were spent for correcting (re-design and re-code)

Class Diagram


TimeStamp
- prodStageName: String
- entry: double
- exit: double
+ setProdStageName(String): void
+ getProdStageName(): String
+ setEntryTime(double): void
+ setExitTime(double): void
+getDuration(): double
Items
- timeStampList: List<TimeStamp>
- entryTime: double
- exitTime: double
+ setEntryTime(double): void
+ setExitTime(double): void
+ stampTime(String): void
+ queueStampTime(String): TimeStamp
+ getPath(): String
BlockState
StarveState
BusyState
Extends
Extends
Extends
States
- name: String
- duration: double
+ incrementDuration(double): void
+ getDuration(): double
+ setName(String): void
+ toString(): String
ProductionStat
- finishedItems: List<Item>
+ add(Item): void
+ getTotalItems(): int
+ getPathsCount: Map<String, Double>
InterStageStorage
- maxQ: int
- q: Queue<Item>
- count: int
- name: String
- qStamp: List<TimeStamp>
- countStamp: List<Integer>
+ enqueue(Item, double): void
+ deque(double): Item
+ isFull(): boolean
+ isEmpty(): boolean
+ getName(): String
+ stampCount(): void
+ getCurrentCount(): int
+ getIsqQueueTime(): double
+ getIsqAvgItemCount(): double
Use
<< abstract >>
ProductionStage
# processItem: Item
# mean: double
# range: double
# r: Random
# events: List<State>
# currentState: State
# prevTime: double
# itemCount: int
# jobSched: Scheduler
+ processItem(double): double <abstract>
+ finishItem(double): void <abstract>
+ calculateProdTime(): double
+ getProdName(): String
+ getTotalTimeOperation(): double
+ incStateDur(double): void
+ setLeftProd(ProductionStage): void
+ setRightProd(ProductionStage): void
+ notifyLeftProd(double): void
+ notifyRightProd(double): void
+ getStageTotalBlocked(): double
+ getStageAvgProduction(): double
+ getStageTotalStarve(): double
PA3
- args: String[]
- prodManager: ProductionLineManager
- mean: double
- range: double
- maxQ: int
+ main(String[]): void
- run(): void
ProductionLineManager
- mean: double
- range: double
- maxQ: int
- prodStageList: List<ProductionStage>
- isqList: List<InterStageStorage>
- des_pq: Scheduler
- prodStat: ProductionStat
- MAX_TIME: double
+ startWork(): void
- processPhase(): void
- createProductionStructure(): void
- printStageStat(): void
- printIsqTime() void
- stampAvgItems(): void
- printPathsCount(): void
- printJobContents(): void
- printStatus(): void
Job
- duration: double
- remainingDuration: double
- stage: ProductionStage
+ getDuration(): double
+ getRemainingDuration(): double
+ finishJob(double): void
+ getStage(): ProductionStage
+ updateRemainingDuration(double): void
Use
Extends
Extends
Extends
Scheduler
- jonQueue: Queue<Job>
- curTime: double
+ addNewJob(double, ProductionStage): void
+ performNextJob(): ProductionStage
+ getCurrentTime(): double
+ getPrintFormat(): String
+ getContents(): String
+ getJobCount(): int
JobComparator
+ compare(Job, Job): int
StartStage
- output: InterStageStorage
+ processItem(double): double
+ finishItem(double): void
MidStage
- input: InterStageStorage
- output: InterStageStorage
+ processItem(double): double
+ finishItem(double): void
LastStage
- input: InterStageStorage
- output: ProductionStat
+ processItem(double): double
+ finishItem(double): void

Inheritance and Polymorphism

Classes that has Inheritance are ProductionStage class and State class. ProductionStage was set as an abstract class and primarily requires extending classes to override the processItem and finishItem because each types of ProductionStage (first, mid, last) have their own way to process items as well as different ways to finish items especially their unblock, unstarve, enque and deque mechanisms. Because of this, ProductionStage can polymorph as either FirstStage, MidStage or LastStage with their own implementation of the said functions.

State Class gets inherited by BusyState, StarveState and BlockState. This was done instead of having flags to take advantage of Java's "instanceof" keyword. This was also done with the possibility of implementing a State Machine Design Pattern, where the functions of ProductionStages(processItem, finishItem) would be inside the states instead of the ProductionStage.

Handling Production Line with Different Topology

By design, the current implementation is agnostic of the topology of the production line. Currently the production line is created manually. For modular production line topology, the structure can be put in an XML document. Additional functionalities required would be to create the parser for the XML file input. Additional functionalities are also required to determine which stages attach to which inter stage storage. Also, automating what production stage is on the left or on the right can be made just by using the inter stage storage. Just by adding this functionality of a file input processor, the code will be able to handle any topologies.

Handling of Items that has "Required Inputs"

There will be more complexity inside the Item class, InterStageStorage class as well as the ProductionStage. The Item class would have a list of requirements, depending on the type of Item it is. The ProductionStage class would always check this list of required Items before going into BusyState. The ProductionStage would also have a list of Items, instead of the current single Item and would be in BlockState whenever an Input item's required items are not yet received. InterStageClass would also have a list of "conveyor belts" for all of the types of possible Items. This will make the production line quite flexible in terms of ProductionStage classes entering block states while waiting for the right input.

In the end, If this were known from the start, I would be changing a couple of functions, replacing data structures. However, the current overall design would still be carried over, as this type of problems can be solved with proper object orientation, with classes with strong cohesion and low coupling.