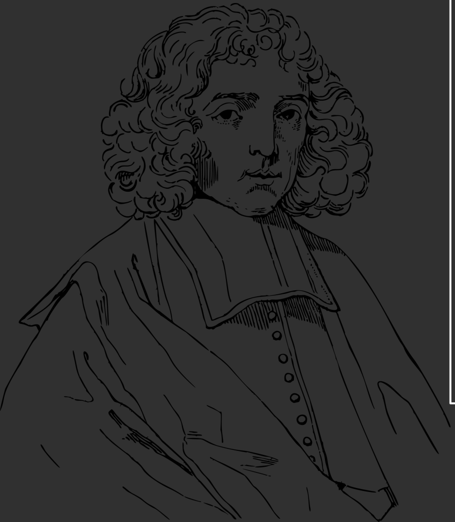
Dining Philosophers

Jaden Bathon and Jacob Grosse

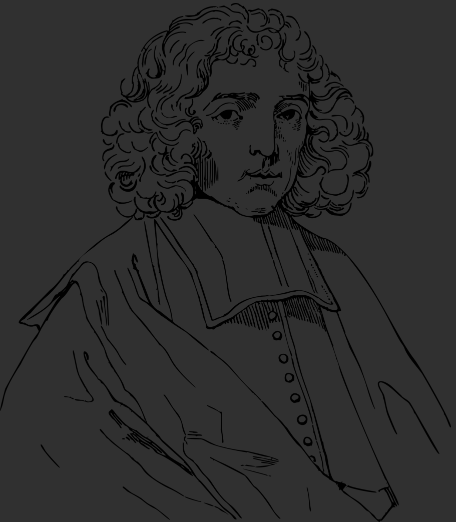
Project Description

The Dining Philosophers problem is a classic problem in the world of computer science and, in a broader sense, logic. The task is to seat a certain number of philosophers at a dinner table with a corresponding number of chopsticks so that each may have only one chopstick. These philosophers must share chopsticks when they become hungry; however, no two of them can pick up the same chopstick at the same time. We were tasked with coding a solution to this problem which allowed independent threads to act as philosophers and effectively share chopsticks amongst themselves to eat rice from the table when they became hungry.

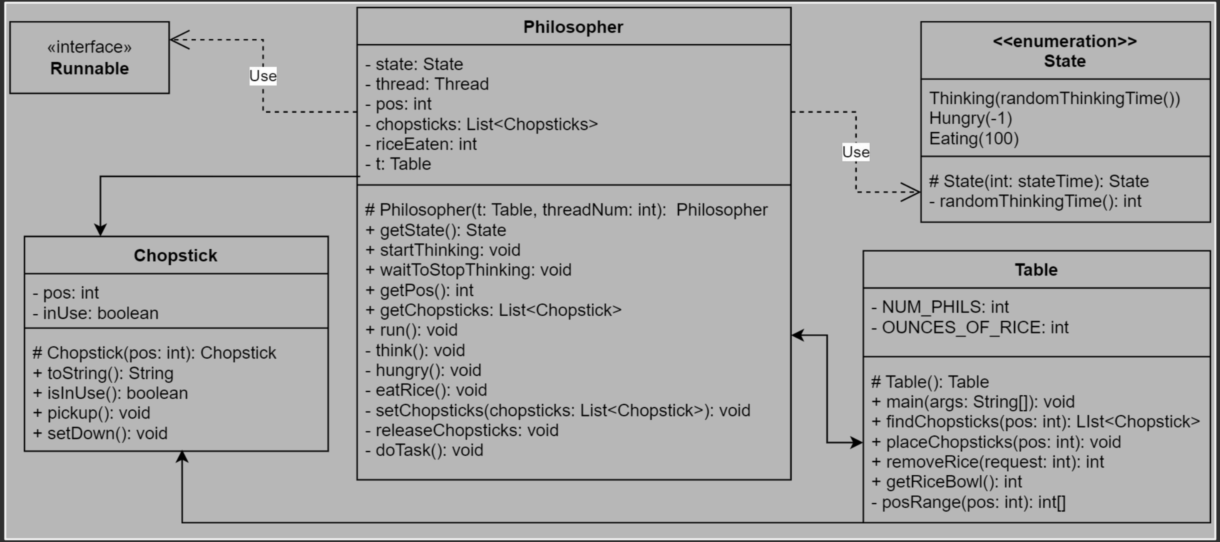


Project Requirements

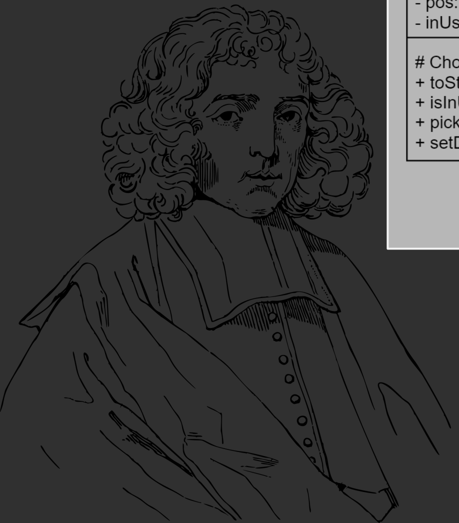
- Uploaded to GitHub
- Runs with ANT
- Solution to the Problem
- Solution is Documented
- Effective Presentation



Project UML Diagram



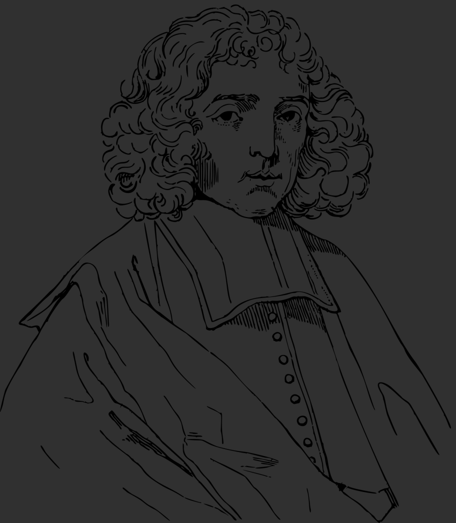
«interface»
Runnable
Use
Philosopher
- state: State
- thread: Thread
- pos: int
- chopsticks: List<Chopsticks>
- riceEaten: int
- t: Table
# Philosopher(t: Table, threadNum: int): Philosopher
+ getState(): State
+ startThinking: void
+ waitToStopThinking: void
+ getPos(): int
+ getChopsticks: List<Chopstick>
+ run(): void
- think(): void
- hungry(): void
- eatRice(): void
- setChopsticks(chopsticks: List<Chopstick>): void
- releaseChopsticks: void
- doTask(): void
Use
<<enumeration>>
State
Thinking(randomThinkingTime())
Hungry(-1)
Eating(100)
# State(int: stateTime): State
- randomThinkingTime(): int
Chopstick
- pos: int
- inUse: boolean
# Chopstick(pos: int): Chopstick
+ toString(): String
+ isInUse(): boolean
+ pickup(): void
+ setDown(): void
Table
- NUM_PHILS: int
- OUNCES_OF_RICE: int
# Table(): Table
+ main(args: String[]): void
+ findChopsticks(pos: int): LIst<Chopstick>
+ placeChopsticks(pos: int): void
+ removeRice(request: int): int
+ getRiceBowl(): int
- posRange(pos: int): int[]



Code Solution

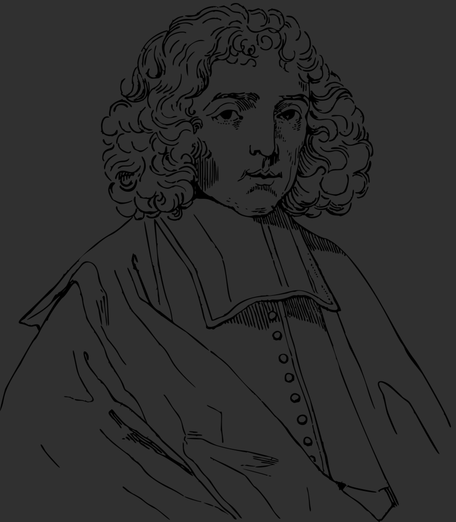
Table.java

- Manages the Rice Bowl.
- Creates and Houses Philosophers and Chopsticks
- Helps Philosophers find Chopsticks



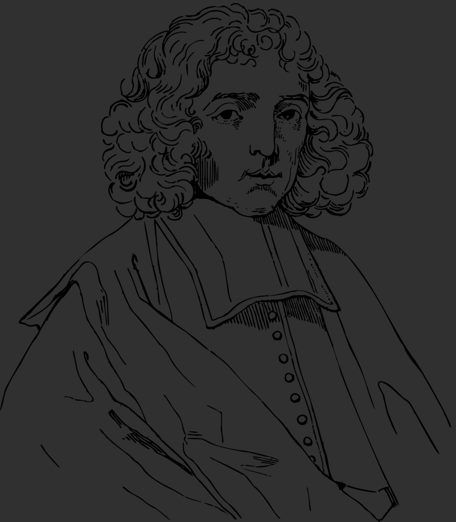
Philosopher.java

- Runs on a Thread
- Sits at a Table
- Knows Where it Sits
- Can Think, Be Hungry, or Eat
- May Be Given Chopsticks
- May Return Chopsticks



Chopstick.java

- Knows Where it Rests
- Knows if it is Being Used
- Can Be Picked Up
    - Yells at you if it already has been picked up…
        - They’re very loud.
            - So much so that the philosopher will put the chopstick back down…
- Can Be Placed Back On the Table



Code Necessary for Each Philosopher Being a Thread

- Synchronization Any Time A Chopstick is Interacted With
- Synchronization On the Bowl of Rice On the Table
- Sleeping Threads to Do Work
- Utilization of wait() & notifyAll() to Schedule Interactions

Challenge Numero Uno

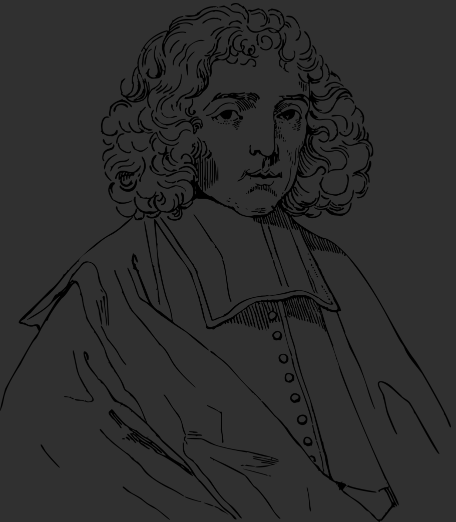
The first challenge was to ensure the philosophers would not pick up a chopstick which was already being used. We managed to overcome this with the Chopstick class which throws an exception if someone attempts to pick up the chopstick while it is in use. This error is handled in the Table class.

Challenge 2 — Educational Haikubaru

Our next challenge was ensuring that philosophers would wait to pick up the chopsticks until one was placed down on the table after being used. This momentary lapse in memory was resolved by telling a philosopher with no valid chopsticks to grab to wait along with the table notifying all philosophers when a chopstick is placed back down on the table.

Challenge 3

Our last challenge was an unnecessary one. We could not get the program to exit when our rice bowl was empty. This was resolved with another frustratingly simple fix. We simply changed the condition on our Philosopher run loop to check the amount of rice remaining in the rice bowl.



Line by Line Code Time! Yay!

Summary

Questions?