1. List of Working Functionality

Description	Status	Note
Delivery/Farmer Priority in Enclosure	**Working**	Delivery always has the priority over the Farmer in the Enclosure
End-of-Simulation Metrics	**Working**	At the end of Simulation, the key metrics are printed out
Farmer Breaks	**Working**	Farmers take a breaks of set length every set amount of time
Farmer/Buyer Priority in Fields	**Working**	When Field is empty, Farmer has the priority until the Field is stacked, then the Buyer has the priority
Field Capacity	**Working**	Fields have randomised capacity in the set range
Logging	**Working**	N/A
Modifiable Parameters	**Working**	N/A
Multiple Buyers	**Working**	N/A
Multiple Farmers	**Working**	N/A
Priority Stocking of Fields	**Working**	Farmers are able to see which Fields need stocking more urgently and pick animals from Enclosure based on that
Randomised Delivery	**Working**	Delivery contains a random number of types of animals and random amount of these animals

2. Division of Work

Liucija Paulina Adomaviciute	Nino Candrlic
Design of classes and state machine	Design of classes and state machine
Implementation of buy() in Buyer.java	Implementation of Buyer multithreading
Implementation of Delivery	Implementation of Farmer multithreading
Implementation of end-of-simulation metrics	Implementation of smart Farmer behaviour and Field urgency
Implementation of Farmer breaks	Implementation of Locks in Field
Implementation of probabalistic spawning of Buyers and Deliveries	Implementation of random Simulation seeds
Implementation of Semaphores in Enclosure	Implementation of thread termination

Implementation of state machine for Farmer	Logging

3. Running the Code

3.1 Parameters

Parameter	**Description**
NUMBER_OF_FARMERS	Number of Farmer threads generated by the Simulation
FARMER_CAPACITY	The amount of animals each Farmer can carry
FARMER_BREAK_TICKS	After how many ticks a Farmer takes a break
FARMER_BREAK_DURATION	For how long a Farmer takes a break (in ticks)
BUYER_SPAWN_TICKS	After how many ticks (on average) a Buyer spawns
DELIVERY_SPAWN_TICKS	After how many ticks (on average) a Delivery spawns
DELIVERY_SIZE	How many animals are generated in a Delivery in total
DAY_DURATION	How long a day is in ticks
MS_PER_TICK	How many milliseconds are in a tick
FIELD_CAPACITY_MAX	The maximum capacity of a Field (exclusive)
FIELD_CAPACITY_MIN	The minimum capacity of a Field (inclusive)
FIELD_STARTING_COUNT	The starting amount of animals in a Field
RNG_SEED	The Simulation seed
RNG_GENERATE_SEED	True if each run of a Simulation generates a new seed, False if the seed set in RNG_SEED is to be used
ANIMAL_TYPES	The types of animals to be used in a Simulation

3.2 Compiling and Running the Code

To compile the code, run “javac –release 21 *.java” command.

To run the code, run “java Simulation” command.

4. Considered Tasks

4.1 Farmer

The Farmer has two main tasks: taking animals from the Enclosure and stocking them in appropriate Fields. We designed a state machine (Figure 1) to represent the Farmer performing these tasks.

At the Enclosure, the Farmer first checks if the conditions for entering the Enclosure are true: the Enclosure isn't empty, no Deliveries are waiting, and there is no Farmer inside. Once the Farmer thread "enters" the Enclosure and acquires the Semaphore, they fetch the Urgency Score (US) from the Simulation and decide which animal(s) to take from the Enclosure.

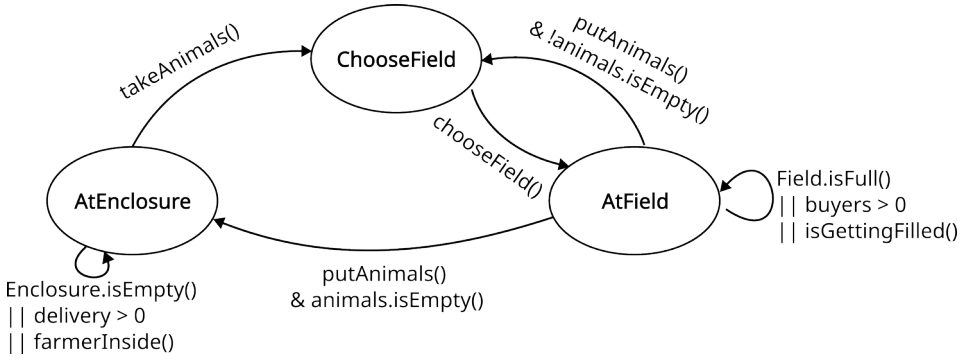

Fig. 1 Farmer state machine

Once the animals have been taken, the Farmer chooses a type of Field based on the animal type it has the most to minimise the time spent "walking" between Fields. At the Field, the thread checks if the Field is full, if Buyers are waiting in the queue, or if there is a Farmer inside. If not, the Farmer acquires the lock and puts the animals into the Field.
If the Farmer still has animals left, the thread chooses a Field again. If not, it returns to the Enclosure.

4.2 Buyer

The Buyer has one task – to take (buy) an animal from a Field. When a Buyer thread spawns, it randomly decides which type of animal it wants to buy. The thread first checks if it can "enter" the Field (the Field isn't empty, and there is no Farmer or Buyer thread already inside) and, if the conditions are met, tries to acquire the Field lock. When the lock is acquired, the Buyer subtracts one from the Field's animal count. Once the task is completed, the Buyer thread releases the lock and terminates.

4.3 Enclosure and Field

The Enclosure and Field objects both act as storage for animals and as access controllers. Their only task is to ensure that only one thread can modify their contents at a time.

5. Patterns & Strategies

During the initial planning and design of classes, it was noted that Farmer and Buyer interactions with the Field resemble the bounded buffer problem - the Farmers being the producers, the Buyers the consumers, and the Field the bounded buffer. The implemented solution uses two binary semaphores and a reentrant lock (mutex lock with conditions). After a Buyer selects a Field, it checks if the thread can enter the Field: the Field is not empty, and the flag for stocking is unset. If the conditions for entry are met, the thread announces its visit by increasing the value of an atomic integer, representing the Buyer threads waiting in the queue, and attempts to acquire the buyerQueue semaphore. If the semaphore is unavailable, the thread sleeps until the Field gives it a

signal to wake up (the semaphore is released). When the Buyer has the semaphore, it acquires the mutex lock, which ensures that only one thread can modify the Field's animal count at a time. The thread decrements the animal count of the Field and releases the lock and the semaphore.

When a Farmer thread decides to approach a Field, it first checks if the conditions for entry are met: the Field isn’t full and the Buyer queue is empty, or the Field is empty. If they are, the thread announces itself by increasing an atomic integer representing the Farmer threads waiting in the queue. Then, the thread attempts to acquire the semaphore farmerQueue and, if the semaphore is already taken, sleeps until the wake-up signal. Once the Farmer has acquired the semaphore, it takes the mutex lock and modifies the contents of the Field.

It was noted that Delivery and Farmer threads interactions with the Enclosure are not a bounded buffer problem, as the Enclosure does not have a capacity. Instead, it was determined to be a simple concurrency problem to avoid race conditions and nondeterministic outcomes. To solve it, two binary semaphores (acting as mutex locks) were implemented in the Enclosure class – one for Farmer (farmerLock) and one for Deliveries (deliveryLock). When a Delivery is generated, it first “queues” itself by increasing the counter in the Enclosure. The delivery counter is an atomic integer in a synchronized method to ensure that no two threads try to modify its value at the same time. Once the Delivery is “queued”, it tries to acquire deliveryLock semaphore. If it can’t do that, the thread sleeps until the semaphore is released, a wake-up signal is received and attempts acquittal again. When the Delivery has the first semaphore, it attempts to take the farmerLock semaphore. When both are acquired, the Delivery modifies the contents of the Enclosure (puts the animals in) and releases both semaphores. When the Farmer thread decides to enter the Enclosure, it first checks if there are any Deliveries queued up. If there are none and the Enclosure is not empty, the thread attempts to acquire the farmerLock, sleeping if the semaphore is not available until the wake-up signal is given. When the Farmer acquires the farmerLock, they modify the contents of the Enclosure (take the animals out) and release the semaphore. A sample semaphore exchange between Delivery, Enclosure and Farmer is illustrated in Figure 2.

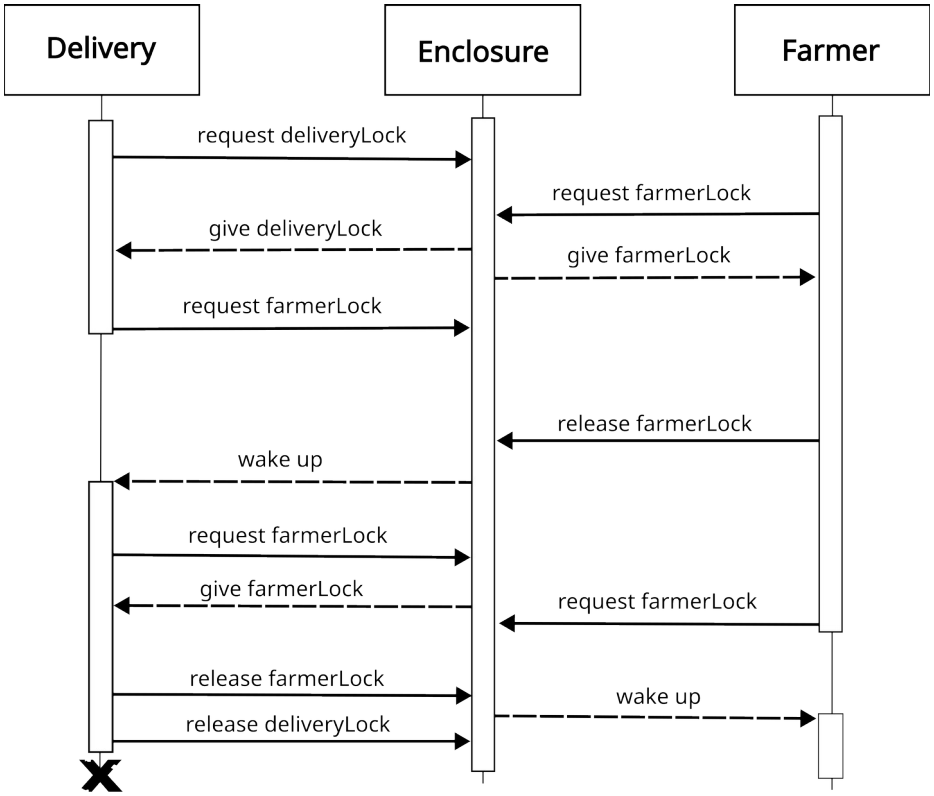

Figure 2. Lock exchange for Enclosure

To prevent deadlock, the Farmer does not request the lock, unless the conditions to take the animals from the Enclosure are met (the Enclosure is not empty or no Deliveries are waiting). This ensures that the Farmer thread is always able to complete its critical section, eliminating the “hold and wait” condition for deadlock. In addition, even if both solutions to the raised problems use multiple locks, there is no circular wait, as only one lock is shared between the different types of threads.

6. Fairness & Starvation

To ensure fairness and prevent thread starvation, multiple strategies were used. First, since the Farmer needs animals to be in the Enclosure to take them out, Delivery always has priority. By prioritising Delivery, the time of Enclosure being empty is minimised, preventing Farmer thread starvation. The second strategy was to utilise Java’s Semaphore class fairness parameter. All semaphores have been initialized with the fairness parameter set to “true”. According to Java’s JDK21 documentation[1], the semaphores queue threads that have requested a permit in the FIFO queue and distribute permits (the locks) based on the order of entry. This way, for example, when multiple Farmer threads request farmerLock, they all get a chance to access the resources (animals) in the Enclosure.

In the bounded-buffer problem (the Buyer and Farmer interaction with the Field), the main issue is fairness. Fairness has to be dynamic, as the status of the Field (empty or full) changes throughout the simulation. That’s why, when a Buyer takes away an animal if the Field's animal count is 0 after the transaction, the priority is given to the Farmers. This ensures that when the Field is empty, the

[1]‘Semaphore (Java SE 21 & JDK 21)’. Oracle, 2023. Accessed: Mar. 13, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/Semaphore.html

Farmers can set a flag preventing Buyers from accessing the Field and stock it up quickly, which minimises the Buyer thread waiting time. Similarly, if the Field is full after a Farmer thread stocks it, the Farmer passes the priority to the Buyers. This solution ensures that the Buyers can remove animals from the Field so that the Farmers can put their animals in it later, allowing all threads to execute and preventing thread starvation.