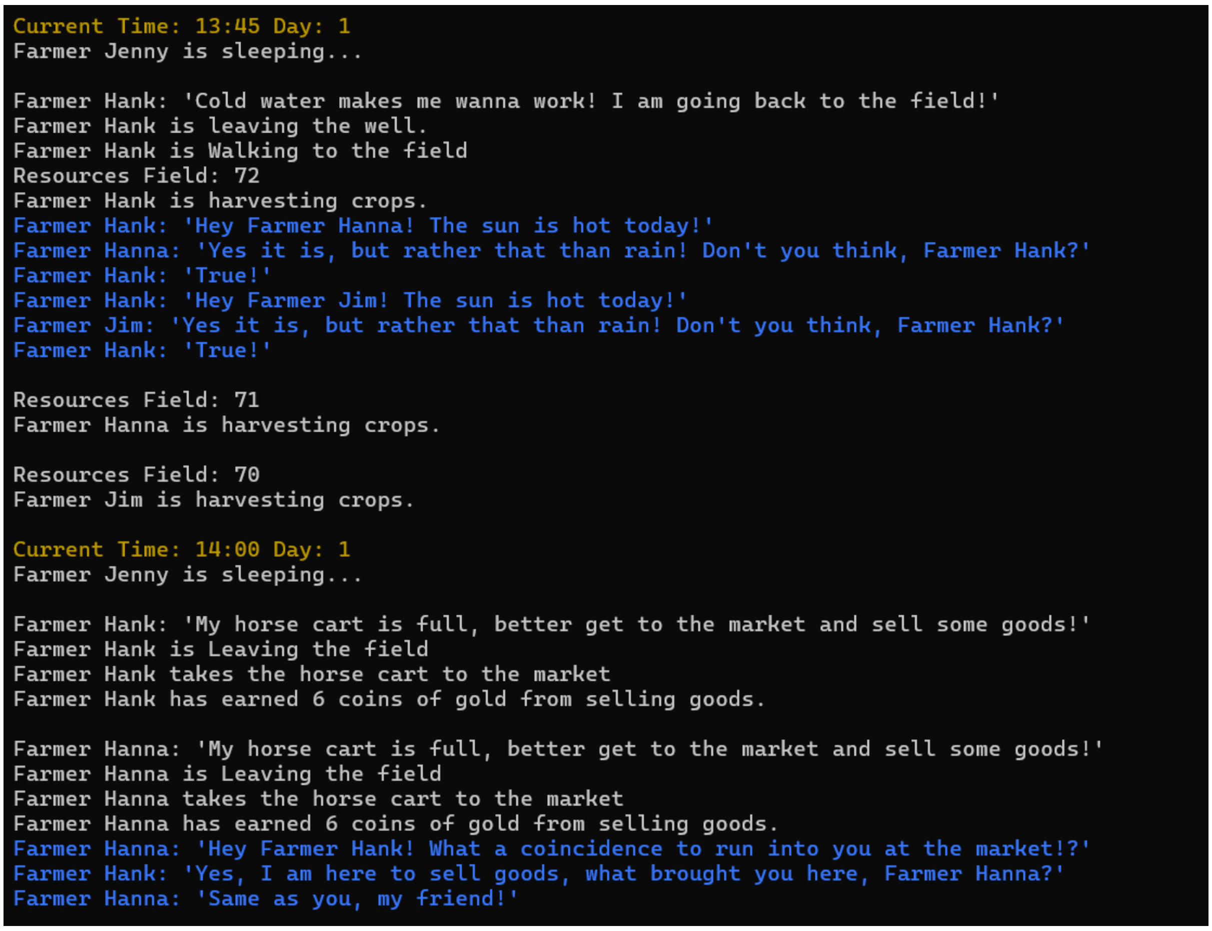
S0011E, Game engine architecture

Assignment 1 - FSM



```
Current Time: 13:45 Day: 1
Farmer Jenny is sleeping...

Farmer Hank: 'Cold water makes me wanna work! I am going back to the field!'
Farmer Hank is leaving the well.
Farmer Hank is Walking to the field
Resources Field: 72
Farmer Hank is harvesting crops.
Farmer Hank: 'Hey Farmer Hanna! The sun is hot today!'
Farmer Hanna: 'Yes it is, but rather that than rain! Don't you think, Farmer Hank?'
Farmer Hank: 'True!'
Farmer Hank: 'Hey Farmer Jim! The sun is hot today!'
Farmer Jim: 'Yes it is, but rather that than rain! Don't you think, Farmer Hank?'
Farmer Hank: 'True!'

Resources Field: 71
Farmer Hanna is harvesting crops.

Resources Field: 70
Farmer Jim is harvesting crops.

Current Time: 14:00 Day: 1
Farmer Jenny is sleeping...

Farmer Hank: 'My horse cart is full, better get to the market and sell some goods!'
Farmer Hank is Leaving the field
Farmer Hank takes the horse cart to the market
Farmer Hank has earned 6 coins of gold from selling goods.

Farmer Hanna: 'My horse cart is full, better get to the market and sell some goods!'
Farmer Hanna is Leaving the field
Farmer Hanna takes the horse cart to the market
Farmer Hanna has earned 6 coins of gold from selling goods.
Farmer Hanna: 'Hey Farmer Hank! What a coincidence to run into you at the market!?'
Farmer Hank: 'Yes, I am here to sell goods, what brought you here, Farmer Hanna?'
Farmer Hanna: 'Same as you, my friend!'
```

Sara Nordström
asaonz-2@student.ltu.se

Source Code Location: https://github.com/unkbitz/FSM
Date: 2025-02-04

Table of Contents

1. Problem Description

The task was to implement a finite state machine (FSM) for a game character. It should be possible to have at least 4 characters simultaneously in the simulation. And the characters should have similar life consisting of:

- Eating and drinking
- Working and earning money
- Having fun with friends
- Sleeping
- communicate by:
  - messaging to plan for a meeting
  - or talking if they happen to be in the same location

For this to work some kind of realistic time management was needed. The lives of the characters should be visualized in some way, either graphically or as text. The communication should also be visualized. The characters should be able to die from starvation or dehydration.

This should be solved in a general way, making the FSM re-usable.

For grade 3 all the above is expected and that is what I decided to aim for.

2. User Guide

2.1 Downloading/cloning repository

The project is located here: https://github.com/unkbitz/FSM

You can either clone the repository or download it as a zip. If you are unfamiliar with git you can choose to download instead.

Downloading the Repository as a ZIP File:

Navigate to the Repository Page:
Go to https://github.com/unkbitz/FSM.

Download the ZIP File:
Click on the green "Code" button located above the list of files.
In the dropdown menu, select "Download ZIP."

Extract the ZIP File:
Once downloaded, extract the contents to your desired location using your operating system's file extraction tools.

2.2 Running the simulation

At location Assignment1\x64\Release within the project folder you created there is a file named “Assignment1.exe”, this is an executable and to run the simulation you simply have to double-click it.

When the simulation runs you can press 1, 2 or 3 to change the speed of the simulation, the higher the number the faster it goes. And space to pause and see some of the current attributes for the living characters.

3. Algorithm Description

3.1 Finite State Machine (FSM):

Each character will operate in the simulation based on the FSM, their behaviour is determined by their current state. Transitions between states occur based on either internal conditions (ex. hunger-, thirst- or energy levels) or external conditions (ex. if it is night time or if the location is out of resources). This design makes the characters keep on operating in simulation, hopefully for a long time if it’s well balanced. The characters will autonomously move between states and uphold a life simulation. through the FSM this is done in a structured way.

To create the different possible states a state factory is used. Instead of creating state instances directly throughout the code, the factory gives a centralized method to get the required state. The State Factory controls streamlining of the state objects in the FSM, ensuring that state transitions are handled efficiently and that the codebase remains modular and maintainable. To manage what transitions to make a state transition table is used.

3.2 Game Loop:

The game loop is an iteration cycle. It processes user input (very little in this simulation), it updates the game state and outputs the simulation. The loop also manages time progression, this will make sure character actions and state transitions happen in a realistic way.

3.3 Messaging System:

Characters communicate in two ways.
If they change location and some other character is at the new location and hasn’t been greeted since he/she came there the characters will exchange a few words.

The second way of communication is sending messages. For this there is a messaging system where characters can send invitations, answer to invitations, notify each other if they can’t come as agreed and answer to such a notice. This system, where messages are queued and processed, makes it easier for the characters to schedule meetings or inform others of changes in plans.

3.4 Resource Management:

There are resources to be managed, basically two types.

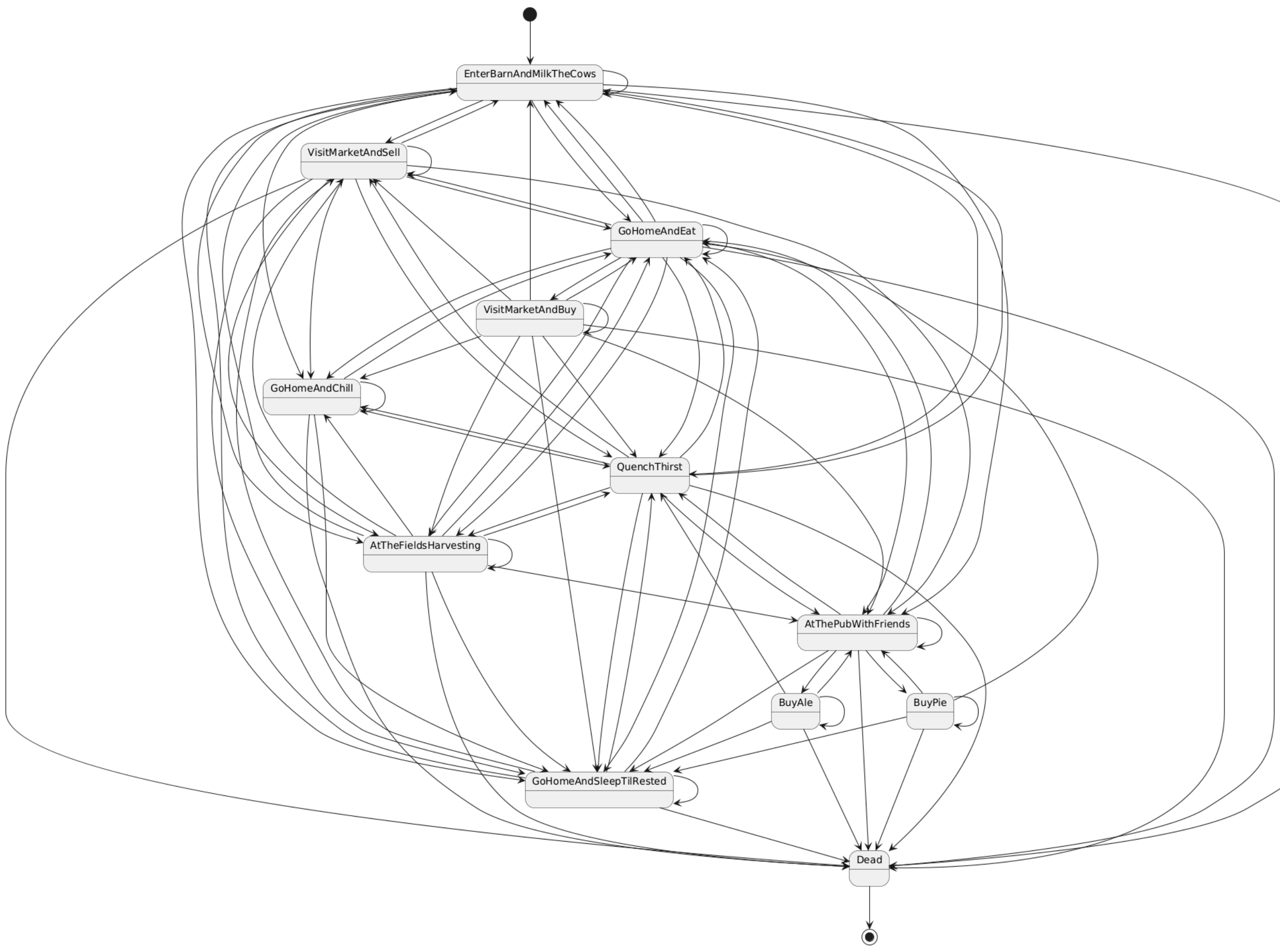
First there is the characters own resources, energy, thirst and hunger. Algorithms monitor these and trigger state transitions through events happening when certain thresholds are reached. Ex. A character will transfer to the “GoHomeAndSleepTilRested” state when energy has dropped to a specific level. If the character fails to balance these resources, death may strike him/her.

The other type of resource is that of the locations, The field has crops, the barn (where they keep cows) have milk and the cottage has food. These are being tracked, decreased and refilled depending on the actions of the characters. Ex. The characters live in a cottage and if they are out of food in the cottage they must buy more, if they can afford it, or die from starvation.

3.5 Event Handling:

The system processes events that affect character states and behaviors. This includes handling environmental changes that may influence character decisions and state transitions.

These algorithms work together to create a dynamic simulation where characters autonomously manage their needs and interact with each other within the simulation environment.



Picture 1. A flow chart over the state transitions in the simulation

4. System Description

4.1 Relationships:

State is a base class and the following classes in my project inherits from State:
AtTheFieldsHarvesting

- AtThePubWithFriends
- BuyAle
- BuyPie
- Dead
- EnterBarnAndMilkTheCows
- GoHomeAndChill
- GoHomeAndEat
- GoHomeAndSleepTilRested
- QuenchThirst
- VisitMarketAndBuy
- VisitMarketAndSell

BaseEntity is a base class and Farmer inherits from it.

Farmer owns StateMachine (composition), meaning that the StateMachine is created within the Farmer class and destroyed when the farmer object is destroyed, therefore the StateMachine can not exist without the Farmer.

Farmer depends on StateFactory meaning that it uses StateFactory to get new states but does not own it.

Farmer depends on GameTime, meaning that it uses GameTime to check the time (e.g., deciding when to sleep or go to the pub), but does not store it.

Farmer is associated with Location, meaning that it has a reference to Location (e.g., barn, market, cottage…) and can move between them; it doesn't own the Locations but rather references to the Locations.

StateFactory depends on State, it creates instances of State but does not own them.
StateFactory also depends on the child classes of State in the same way:

- AtTheFieldsHarvesting
- AtThePubWithFriends
- BuyAle
- BuyPie
- EnterBarnAndMilkTheCows
- GoHomeAndChill
- GoHomeAndEat
- GoHomeAndSleepTilRested
- QuenchThirst
- VisitMarketAndBuy

- VisitMarketAndSell

StateMachine is associated with State, it manages the current State and switches between states.

Summary of the system description:
State is a base class for multiple states.
Farmer follows an FSM pattern, and uses StateMachine to control state transitions.
StateFactory creates the states but does not own states.
Farmer moves between locations but does not own them.
Farmer uses GameTime for time-based decisions.

There is an appendices added with a UML class diagram.

4.2 The class roles:

4.2.1 Class: BaseEntity;
Role: A base class for entities in the game, it gives each entity a unique ID.
Key Methods:
SetID(int value): Ensures that each entity gets a unique ID.

Key Member:
static int m_iNextValidID: Holds the next ID.

4.2.2 Class: Farmer:
Role: Represents a farmer entity that moves between locations, manages resources, and transitions between states.

Key Methods:
Update(GameTime gameTime):
Determines current state transitions based on game time and farmer conditions (hunger, thirst, tiredness). Calls ChangeState() if a state transition is needed.

ChangeState(State<Farmer>* pNewState):
Handles state transitions, ensuring an exit from the previous state and entry into the new state.

GetNextState(const std::string& currentState, const std::string& event):
Uses m_stateTransitionTable to find the next state based on the current state and an event.

ChangeLocation(Location* newLocation):
Moves the farmer to a new location, updating references.

SendMessage(Farmer& recipient, const std::string& message):
Makes communication between farmers possible by using a message queue.

ProcessMessages(std::vector<std::unique_ptr<Farmer>>& farmers):
Handles incoming messages and decides responses.

4.2.3 Class: StateFactory:
Role: Provides a centralized way to create and retrieve state instances for the Farmer.Returns existing instances of states instead of creating new ones.

Key Methods:
State<Farmer>* GetState(const std::string& stateName):
Takes a string (name) as input and Returns an instance of the requested state.

4.2.4 Class: StateMachine
Role: Manages the state transitions for an entity (Farmer).
Ensures that each state executes correctly before switching to a new one.

Key Methods:
SetCurrentState(State<entity_type>* s):
Sets the initial current state of the entity.

Update() const:
Calls Execute() on the current state, allowing it to perform actions.

ChangeState(State<entity_type>* pNewState):
Exits the current state, switches to a new state, and enters the new state.

4.2.5 Class: State:
Role: Base class (abstract) for defining different states that an entity (Farmer) can be in.
Provides a common interface for all states.

Key Methods (Pure Virtual):
Enter(entity_type* entity):
This happens when entering the new state.

Execute(entity_type* entity):
This is executed repeatedly (every update cycle).

Exit(entity_type* entity, std::string nextState):
This happens when leaving the state.

GetEvent(entity_type* entity):
This method determines what event is set to and the event might trigger a state transition.

4.2.6 Class: Location:
Role: Represents physical locations in the simulation (like Barn, Market, Field).
Stores resource availability at that location (like milk, crops, food).
Has Getters and setters for the resources.
Used by entities (like Farmer) to change locations and interact with resources

5. Limitations of the Solution

One main limitation I can think of is that the agents all say the same things when encountering the same situation, they respond the same way to each other and the conversations are overall very repeated. Making it all very uninteresting to watch.

In this case all the agents are updated every iteration, if they were put into a game it would make sense to only update those needing it. If there were many farmers and not just four, in the game world it could become very inefficient to update them all every frame. But for this assignment when we’re only watching the agents and have a limited small number of agents it’s fine.

The fact that I have hardcoded that some things happen at a certain time may seem unrealistic. It is really undynamic that every day at 12:00 the farmers may invite each other to come to the pub in the evening. And that they go to the pub the same time every day (if they are not finished early with their work). This could probably be done more dynamically. For example they could send invitations when having a break and deciding on a time to be there based on how much work they had left or something like that.

6. Discussion

Just having learnt what a finite state machine is developing one was challenging. On the other hand the learning was huge from not knowing a thing about it to learning how it works, what classes are needed, how they could be connected/interact to create agents transitioning between states driven by events rather than any human interacting with the system to make the agents do stuff.

The first week was spent on reading in the course literature and starting the implementation, I got it to, kind of, work. But as it grew and the different possible states became more and more it became really messy, since I only used if-statements inside each Execute() to determine what was going to happen next. I didn’t have events, or a transition table and I didn’t have the statefactory, instead the creation of the state instances happened all over.

This approach made it all very hard to modify and maintain the code base as it grew.

Week two I realised this was unsustainable and I also remembered from the lectures that if-statements was a bad way of implementing this and a transition table was to prefere. So I started to rebuild basically the entire project.

I also spent a lot of time dealing with what should be in the StateMachine class and what should be in the Farmer class. since they both have Update and ChangeState.

Finally I came to realise that Farmer::Update() should determine what state the farmer should be in, while StateMachine::Update() should execute the logic of the current state. Together, they ensure that the farmer transitions correctly between states and that the current state behavior is continuously executed.

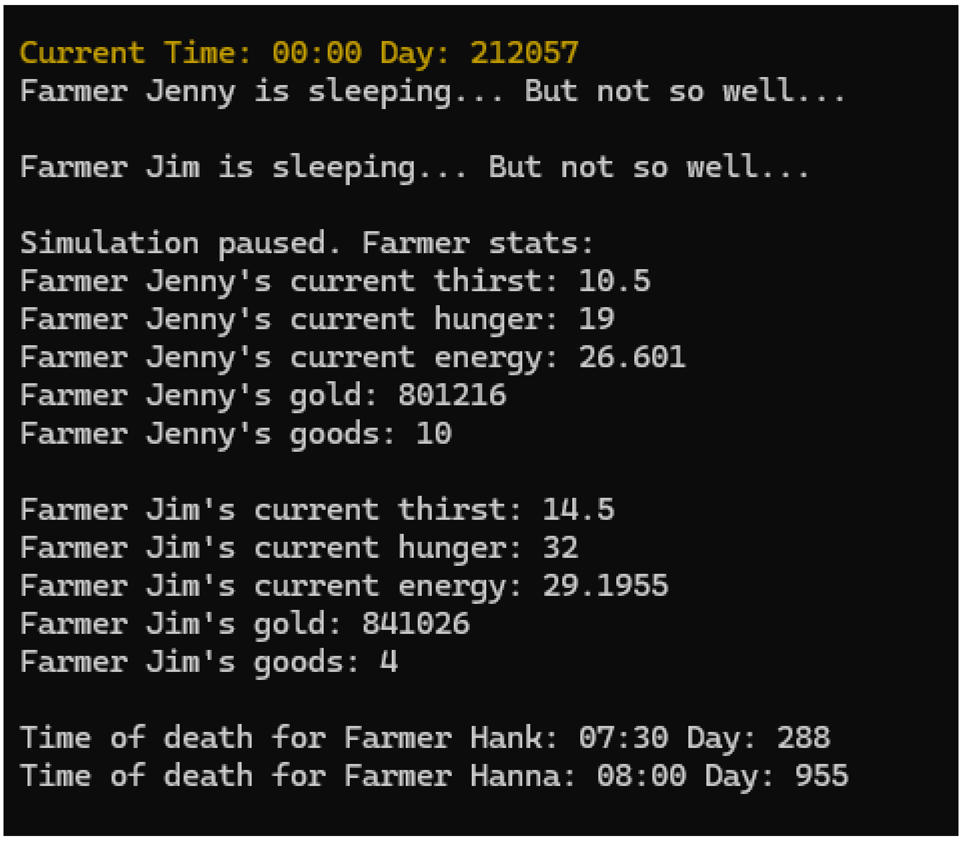
And for ChangeState; Farmer::ChangeState() should ensure valid transitions while StateMachine::ChangeState() should perform the actual transition between states.

Without Farmer::Update(), the farmer would never switch states.
Without StateMachine::Update(), the current state wouldn't execute its logic.
Without Farmer::ChangeState(), state changes might happen unnecessarily or incorrectly.
Without StateMachine::ChangeState(), the actual transition process (exit, enter) wouldn't be executed properly.

At this time I only iterated a certain amount of times and did not have “time” included in the project. So as the transitions started working as expected overall, and as it worked when I added more farmers as well, time was implemented to make it feel more realistic and since sleeping normally happens during the night time and so on. Now the simulation would run until all farmers were dead. And the real balancing of farmer-resources, location-.resources, death, time and so on could start. This took quite a long time, but eventually it came together.

The last I added was the communication between the farmers. I have two different kinds, one is sending messages using a queue and the other is when they happen to meet in person (be in the same location at the same time) they change a few words.
The hardest part of this was understanding where to output these conversations to make it feel logical, for instance the conversations in meetings had to be outputted while they were still at the same location. I tried to add them saying goodbye when leaving a place other farmers were currently at, but due to time pressure I had to abandon that idea. I also wanted to vary the conversations more, but since I got it to work and again due to time pressure I figured it was unnecessary. Other than that this part was quite straight forward.

7. Test Runs



```
Current Time: 00:00 Day: 212057
Farmer Jenny is sleeping... But not so well...

Farmer Jim is sleeping... But not so well...

Simulation paused. Farmer stats:
Farmer Jenny's current thirst: 10.5
Farmer Jenny's current hunger: 19
Farmer Jenny's current energy: 26.601
Farmer Jenny's gold: 801216
Farmer Jenny's goods: 10

Farmer Jim's current thirst: 14.5
Farmer Jim's current hunger: 32
Farmer Jim's current energy: 29.1955
Farmer Jim's gold: 841026
Farmer Jim's goods: 4

Time of death for Farmer Hank: 07:30 Day: 288
Time of death for Farmer Hanna: 08:00 Day: 955
```

Picture 2. output showing the two farmers who died during the final testrun.

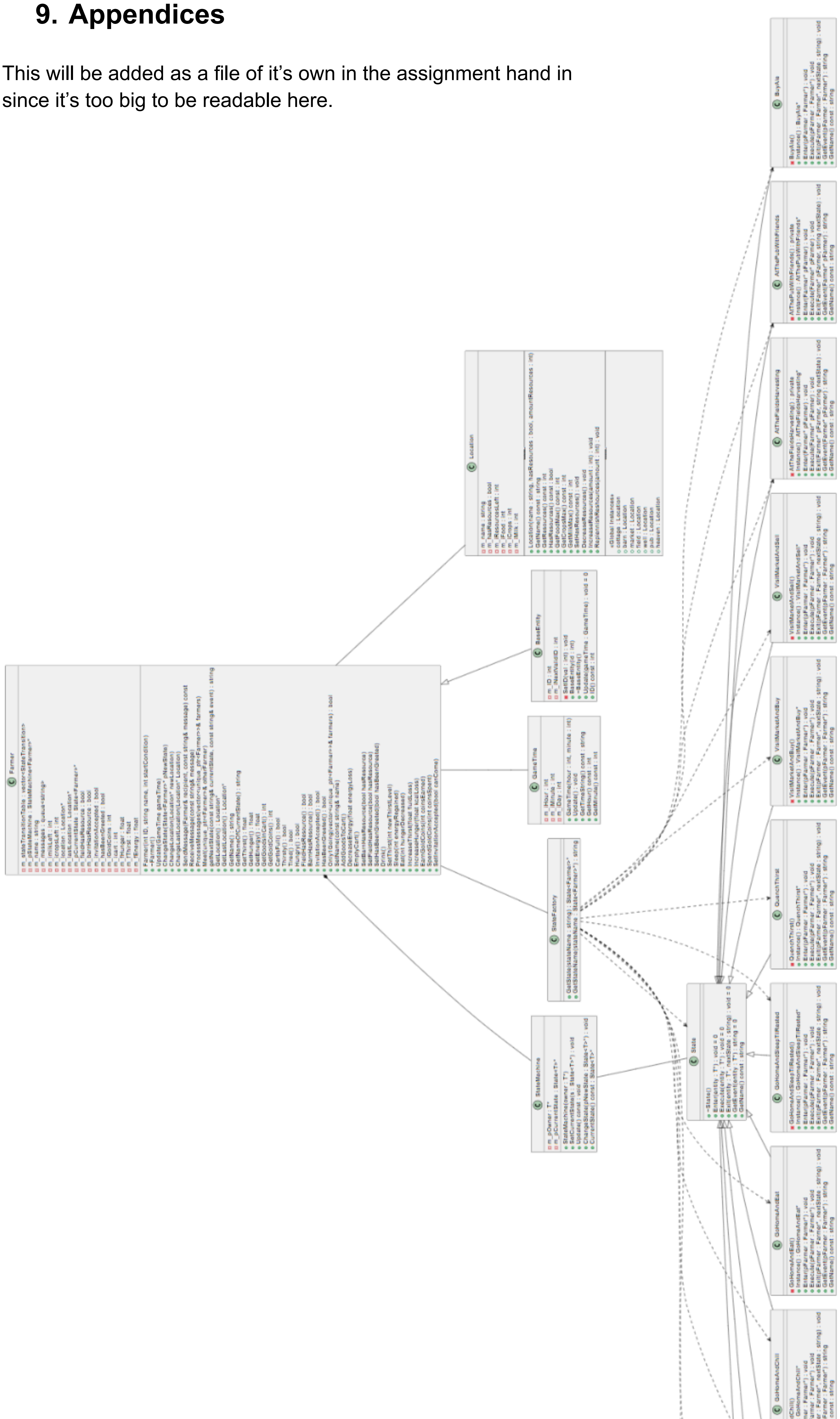
The first farmer died at iteration 27583, the cause was hunger.
The second farmer died at 91713, the cause was
The other two farmers lived for over 20 000 000 iterations and were still alive as I did the hand in. The simulation had been running for 4h and was still going.

8. References

Buckland, M. (2005). *Programming game AI by example.* Wordware Publishing, Inc.

9. Appendices

This will be added as a file of it's own in the assignment hand in since it's too big to be readable here.


Farmer
Location
BaseEntity
GameTime
StateFactory
StateMachine
State
BuyAle
AtThePubWithFriends
AtTheFieldsHarvesting
VisitMarketAndSell
VisitMarketAndBuy
QuenchThirst
GoHomeAndSleepTilRested
GoHomeAndEat
GoHomeAndChill