# “Potion or Poison?” Technical Design Document

Team: Imaan Salie (SLXIMA002), Imaan Sayed (SYDIMA002), Cassandra Wallace (WLLCAS004)

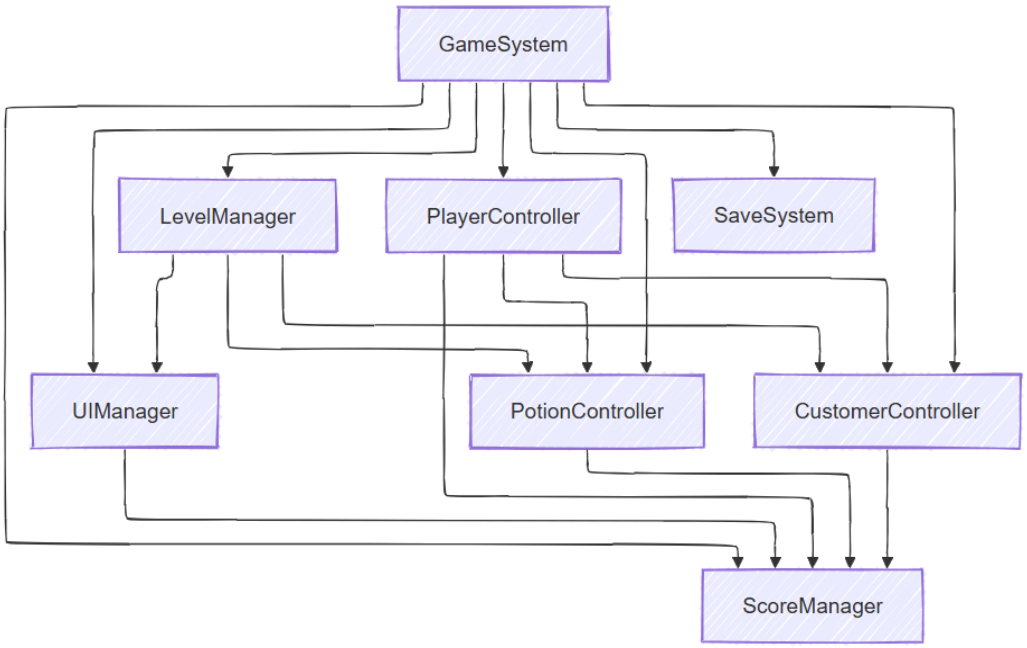
Date: 23 September 2024

Preliminary ideas, which may be revised, are highlighted in red.

## System Design:

The technical architecture of *“Potion or Poison?”* is designed around Unity’s component-based system, emphasising modularity and scalability. The game utilises Unity’s UI and physics systems for the drag-and-drop interaction with the potion ingredients. It is also crucial to have a tracking system for the player’s lives, correctly crafted requests and Threat Level, updated dynamically based on player choices.

The game will be developed using a combination of MonoBehaviour C# scripts to control specific GameObject behaviours, with plain C# classes for system-wide logic, specifically with player interactions and the implementation of the scoring and narrative systems. We aim to follow the Model-View-Controller (MVC) pattern where applicable to separate data, presentation, and logic.




The game’s system components and their points of interaction through scripts are visualised in the above diagram.

## Iterative Development

The system will be developed and tested in a number of iterations:

1. **First iteration:** Design and implement the potion-making mechanics, incorporating a drag-and-drop feature for combining ingredients and animations to illustrate potion production.
2. **Second iteration:** Design and integrate customer interactions, adding animations for visual cues (to indicate imposters) and linking customer requests to the appropriate potions.
3. **Third iteration:** Develop narrative elements, including rebel activity notices provided to the player between levels, and implement the imposter detection mechanic where poison can be added to potions.
4. **Fourth iteration:** Test and improve the game based on user feedback.

## Core Systems and Script Descriptions

The game's systems interact through a combination of direct method calls, events, and the Unity message system.

1. **`GameSystem.cs`:** This system utilises the singleton pattern for global game state management, coordinating the overall game flow. It coordinates all communication between other cores systems, as well as managing level progression. Key responsibilities include initialising other managers and systems, triggering level transitions, and notifying `UIManager` to update UI based on game state changes
2. **`UIManager.cs`:** This system handles all UI elements and transitions, updating based on changes in game state. It particularly manages input events for the player's drag-and-drop cursor functionality. By having this UIManager, it effectively separates the visual and audio presentation from the game's data and logic, in accordance with MVC principles.

   *Key Algorithms:*

   - `UpdateUI(GameState state)`, which updates all UI elements based on the current game state.

- `ShowIntermissionScreen()`, which displays the intermission screen with the player's performance on the previous level.

3. **`PlayerController.cs`:** This script manages the player's input and cursor interactions, which is vital for all core gameplay of the game. Specifically, it handles the implementation of the drag-and-drop functionality using Unity's EventSystem.

   *Key Algorithms:*

   - `OnBeginDrag(PointerEventData eventData)`, which initiates the dragging of an ingredient or potion.
   - `OnDrop(PointerEventData eventData)`, which handles the dropping of ingredients in the brewing oven, the delivering of potions to the customer, etc.

4. **`PotionController.cs`:** This script manages all valid potion combinations and crafting logic. It stores the valid combinations of all ingredients and processes invalid crafting attempts. Ingredients will be efficiently validated via a dictionary to determine valid potion combinations.

   *Key Algorithms:*

   - `CraftPotion(Ingredient firstIngredient, Ingredient secondIngredient)`, which returns the crafted `Potion GameObject`.
   - `PoisonPotion(Potion potion)`, which applies the poison effect to the given `Potion GameObject`.
   - `DiscardPotion(Potion potion)`, which removes the given `Potion GameObject` from the scene..
   - `DeliverPotion(Potion potion, Customer customer)`, which notifies `ScoreManager` of successful or failed potion deliveries, as well if an incorrect/correct rebel poisoning occurred.

5. **`CustomerController.cs`:** This script spawns customer sprites and manages their visual cues and speech bubbles for their potion orders & reasonings. It generates the customer requests to be displayed and manages the per-customer time ("happiness") limits for each customer based on the current level.

We are stretching for randomised, procedural customer generation by using modular customer sprites, with additional randomness for if they are rebels and the corresponding visual cues. This will also enhance the replayability of the game. However, if this causes problems, we can simply still have a randomised order of hard-coded customers, which also allows more freedom as a narrative device, e.g. specific customers with their own overarching storyline that will arrive in the same level every replay.

Either way, this script will also handle the dialogue system for customer orders and reasoning, which is simply a fetching from some look-up table of our written potion orders.

*Key Algorithms:*

- `LoadCustomer(Level level)`, which generates an appropriate `Customer GameObject` for the given level's difficulty.
- `GeneratePotionOrder(LEvel level, Customer customer)`, which creates a potion order for the given `Customer GameObject` for the given level's difficulty.
- `ExitCustomer(customerObject)`, which removes the given `Customer GameObject` from the scene.

6. **`ScoreManager.cs:`** This script tracks the player's number of lives, correctly & incorrectly crafted potions, poisoned rebels, and innocents killed.

   Specifically, it handles the Player's Lives and the Threat Level, updating them based on player actions.

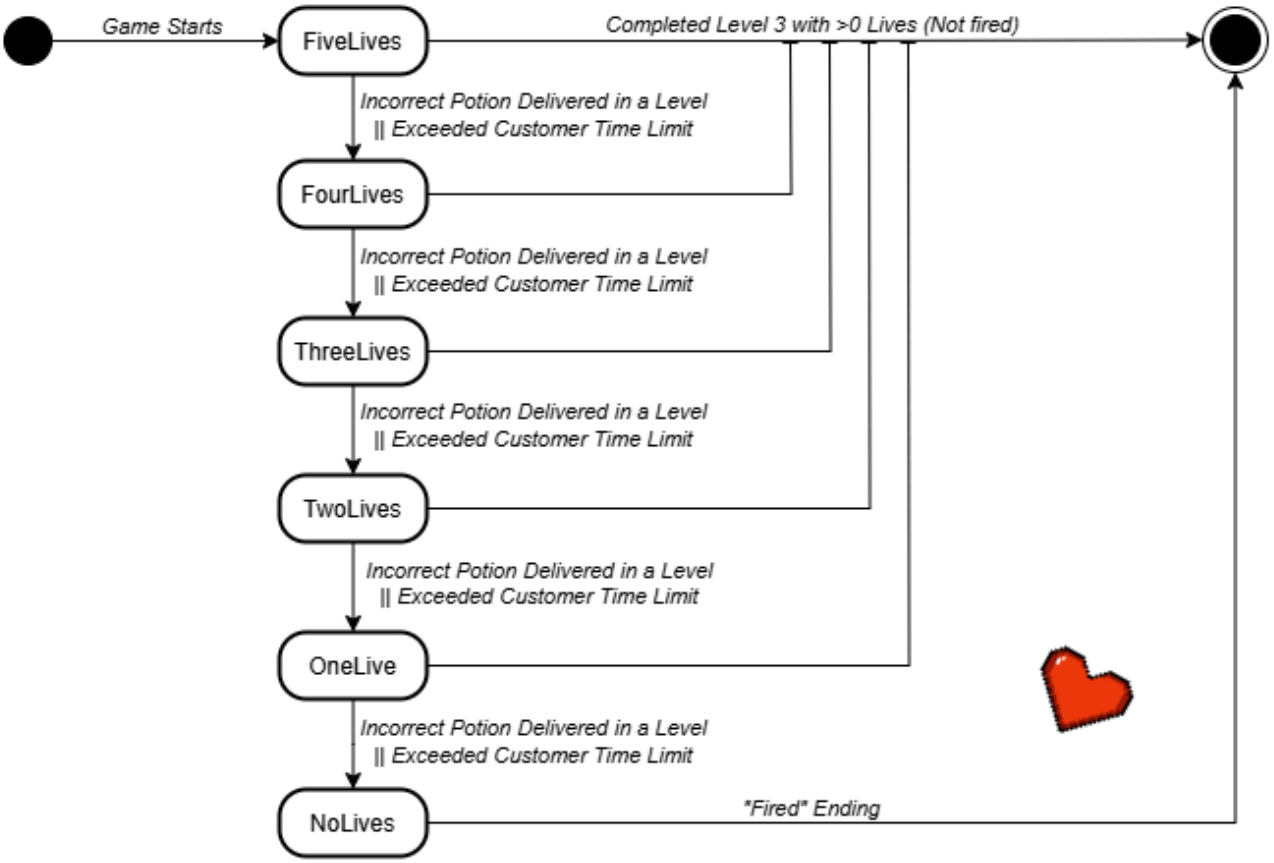
   Every time the player delivers an incorrect potion, or the per-customer time ("happiness") limit runs out, the player loses one of their five lives.

   The Threat Level percentage, starting from 0%, will increase every time the player misses a rebel and does not poison their potion.

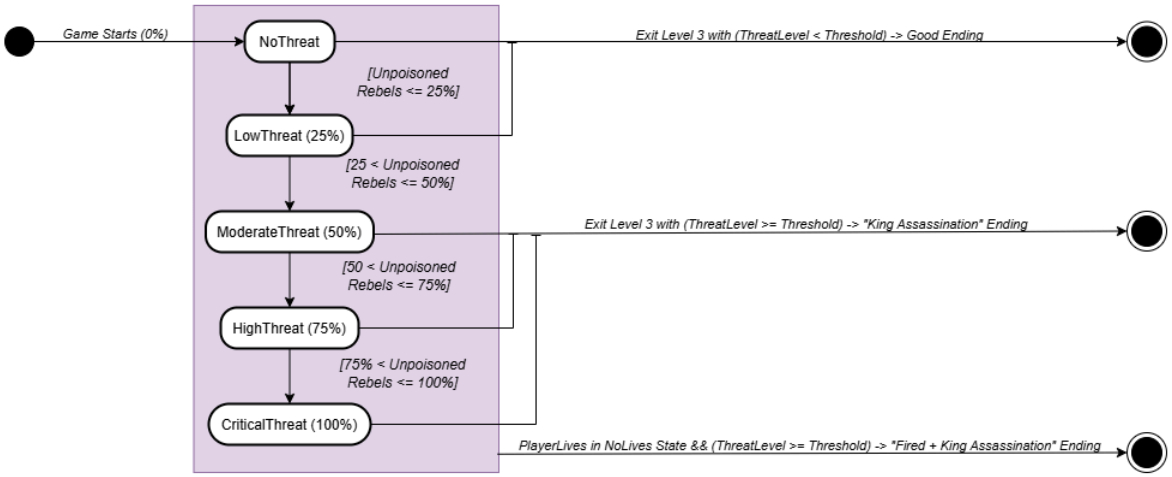
   In this way, these algorithms will act more as helper methods to be invoked by `PotionController` and `ScoreController`, providing the methods for updating and querying game scores. It will also work hand-in-hand by the Level and Narrative System to trigger the corresponding endings.

Key Algorithms:

- `CurrentLives()`, which returns the current number of lives the player has.
- `LoseLife()`, which decreases the player's life count by one.
- `CurrentThreatLevel()`, which returns the current threat level of the world as a result of the player's performance as a float.
- `IncreaseThreatLevel(float amount)`, which increases the current threat level of the world by the given amount.
- `CheckGameOver()`, which checks if the current game should end based on the current number of player lives and threat level.




*State Machine Diagram for the Player's Lives, visualised throughout the game.*




*State Machine Diagram for the Threat Level, visualised throughout the game.*

7. **`LevelManager.cs:`** This script handles the progression between levels, managing the various level-specific initialisations that need to be performed at the beginning of a level, such as the current level's set of available ingredients to be displayed on the shelves, and the current level's per-customer time limit. It also retrieves hard-coded story progression between levels via the intermission newspapers/bulletins.

   As a stretch goal, this script will also handle any dynamic, branching narrative content that we may want to add on the actual newspapers. This will be dependent on how this script could track the player's choices in some data structure.

   Key Algorithms:

   - `NextLevel(Level currentLevel)`, which performs a level transition & initialisation, and notifies `GameSystem`.
   - `GetIntermissionStory(int levelNumber)`, which retrieves the appropriate story content for the intermission screen.

8. **SaveSystem.cs:** This script manages the game save data and player progress, by implementing the serialisation and deserialisation of the game state to remember where the user survived up until, as well as the potions they have learned. This is considered a stretch goal, as we are playing around with the idea of our game being an arcade-type, "permadeath" system, where the player would play a run of our game in one go. This makes sense considering the short, overall length of our game.

   *Key Algorithms:*

- `SaveGame(GameState currentState)`, which serialises and stores the current game state.
- `LoadGame()`, which deserialises and returns the previously saved game state.

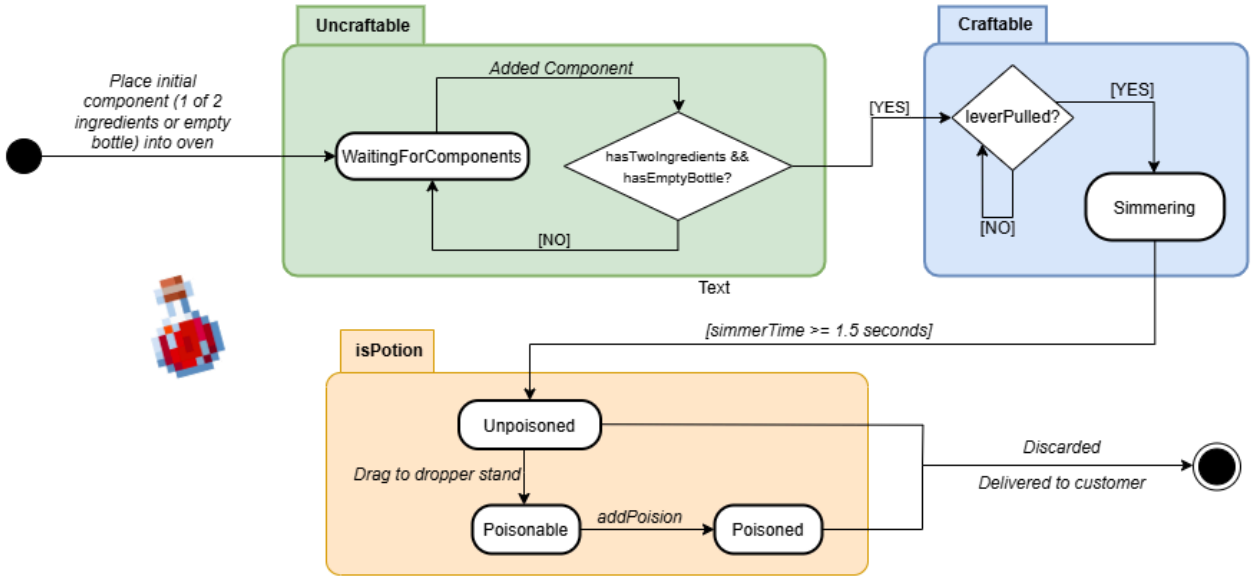
## Key Classes and Interfaces

### Ingredient

- `string Name`
- `Sprite Icon`
- `List<string> Effects`

### Potion

- `string Name`
- `Sprite Icon`
- `List<Ingredient> Ingredients` (length 2)
- `bool IsPoisoned`
- `poison()`




*State Machine Diagram for a PotionOrder, visualising its life cycle within a level.*

**Customer**

- `Sprite Icon`
- `bool IsRebel`
- `Potion RequestedPotion`
- `string PotionReasoning`
- `float happinessLevel`
- `updateHappiness(float delta)` (called every frame update)

**Level**

- `int DayNumber` (in {1,2,3})
- `List<Ingredient> AvailableIngredients`
- `float TimeLimitPerCustomer`

## Art Assets:

Overall, the main art that the game needs will be the 2D sprites for customers, potions, ingredients, and the shop environment.

### UI Elements

There will also be necessary UI elements for the various buttons, metres, and indicators, specifically within the level and the level intermission screen.

### Customer Design

A small set of “base” sprites will be used to simplify customer creation. Base sprites take on slight adjustments according to the level the player is in wherein they will take on subtle characteristics representing their affiliation. For example, if we learn in level two that rebel sympathisers wear red scarves, some of the level three sprites will wear red accessories. *This may be discarded to reward replay, where the player uses information from later levels to inform deduction in earlier ones.*

### Animations

To simplify the large number of interactions while maintain the “aliveness” of the game, animation will focus on simple manipulation of the 2D character models (blinking = opening/closing eyes, sweating = droplet fading in/out, shifty eyes = move pupils left/right) to give impostors suspicious characters and to make all characters seem more lifelike (therefore charming).

For feasibility within the timeline, potions and the potion-making process do not have animations. The exception is to give feedback that the potion is being made (oven door sliding up/down to indicate “crafting” (simmering) is being undergone) and that a potion has been poisoned (ghost skull fading in/out on top of bottle).

## Audio Assets:

### Background Music

To emphasise the charming and cosy feel of the game we will make use of gentle, ambient, fantasy-like music to create an immersive environment for the players.

### Sound Effects

The sound file formats should be .wav, .mp3, .caf or .aif. All sound effects will be less than 30 seconds in duration. The inclusion and type of background music is undecided.

| Resource Type | Duration |
|---|---|
| Ingredient Added | 0.1s - 0.3s |
| Potion brewing | 0.1s - 0.3s |
| Potion successfully made | 0.1s - 0.3s |
| Customer satisfied | 0.1s - 0.3s |
| Customer unsatisfied | 0.1s - 0.3s |
| Ingredient unlocked | 0.1s - 0.3s |
| Potion added to recipe book | 0.1s - 0.3s |
| Poison dropper picked up | 0.1s - 0.3s |
| Poison dropped into potion | 0.1s - 0.3s |
| Game level completed | 1.0s - 3.0s |

### Voice Acting

As a stretch goal, we could utilise some minimal voice acting that would trigger when a customer's potion order speech bubble is displayed. This could be very simple, such as the mumbling, gibberish-style dialogue that is used in Animal Crossing.

## Team Skills, Timeline, and Feasibility:

### Team Skills

- Imaan Sayed [Programming, Audio, Art]
- Imaan Salie [Programming, Audio, Art]
- Cassandra Wallace [Programming, Audio, Storytelling]

### Timeline

*Week 1:* Pitch, Game & Technical Design Documents, Initial Prototype.

*Week 2:* UI Setup, Core Mechanics with Potion Crafting & Customers.

*Week 3:* Narrative Elements, Test & Iterate Based on Playtesting.

*Week 4:* Polish Visuals, Narrative and Audio.

### Feasibility

Given the available time and team skills, the core gameplay mechanics—drag-and-drop potion crafting, customer interaction, and scoring systems—are feasible within the course timeline. The use of Unity's built-in tools and asset store will streamline development, and modular scripting will allow the team to focus on iterative development and playtesting for fine-tuning the game. The game's technical foundation ensures that it is not only feasible to implement, but also leaves room for potential expansion and refinement during development

However, we have identified the following key potential challenges and risks with the game:

1. ***Balancing Difficulty Progression:*** Our mitigation for this is to implement a robust playtesting process. At the time of writing, we have already performed an external playtest with our paper prototype, and some key insights for this have already been gained. We also have the stretch goal to add a flexible difficulty setting, where the user can adjust it as desired. This will primarily vary how long the player has to create a

customer's potion. For example, an expert player who wants an extremely challenging gameplay could maximise the difficulty, and have very fast-paced potion crafting within the level.

2. ***Ensuring Engaging and Coherent Narrative:*** A key distinguishing feature of the game is the interesting aspect of how the player's performance impacts the in-game world. To ensure that this is still fun and engaging, we will perform regular story reviews and iterations, while also seeking feedback from our playtests on the storytelling.

3. ***Optimising Performance for the Satisfying, Smooth Drag-and-Drop Mechanics:*** The dragging-and-dropping mechanic will be extremely crucial to the satisfying feel that we want the player to experience, much like in how "*Paper's, Please"*. If this mechanic feels jittery or clunky, the player's experience of the game will be severely deteriorated. Extensive playtesting and alterations will be performed as needed to fine-tune this aspect, to ensure it feels as desired.

---