# Design Documentation

## Team 81

Andrew Durnford, Eric Jiang, Calvin Luo, Edison Qu, Surajj Vinodh Kumar

# Virtual Pet Game

## Design Documentation Document

### Contributors

| Task / Part | Contributors | Description / Notes |
|---|---|---|
| Main Page & Introduction | Calvin, Edison | Calvin set up the wiki structure and title formatting. Edison created document templates and updated the introduction from the requirements document. |
| Class Diagrams | Surajj, Andrew | Surajj selected the UML diagramming tool and designed the core class structure. Andrew defined the attributes, methods, and documented the relationships between classes. |
| User Interface Mockup | Calvin | Created wireframes for all application screens using Balsamiq, including interactive elements and design rationale documentation. |
| File Formats | Edison, Surajj | Selected CSV as the file format, defined data structures, and documented column organization for all game data files. Edison researched external libraries while Surajj defined the data hierarchy. |
| Development Environment | Andrew | Specified IntelliJ as the IDE, documented external libraries including JUnit 5 and Jackson, and outlined the version control workflow. |
| Patterns | Eric | Researched and documented appropriate design patterns for the application with references and implementation approaches. |
| Summary | Eric, Edison | Eric drafted the summary paragraph while Edison compiled the technical terms glossary. Both worked on proofreading the final document. |

# Table of Contents

# Introduction

## Overview

This design documentation outlines the architectural and interface design for our Virtual Pet Game, transforming the captured requirements into a concrete technical implementation plan. The design emphasizes a modular object-oriented approach that separates game logic from user interface components while maintaining clear relationships between core entities. We've structured the system around key classes such as Player, Pet, Shop, and Game, with well-defined attributes and methods that support the virtual pet care experience. Our user interface design prioritizes intuitive navigation, consistent visual elements, and age-appropriate aesthetics to create an engaging environment for both children and parents. Data persistence is addressed through a structured CSV file system with comprehensive validation, while implementation will leverage established design patterns to ensure maintainability and extensibility.

## Objectives

The primary objectives of this design document are to:

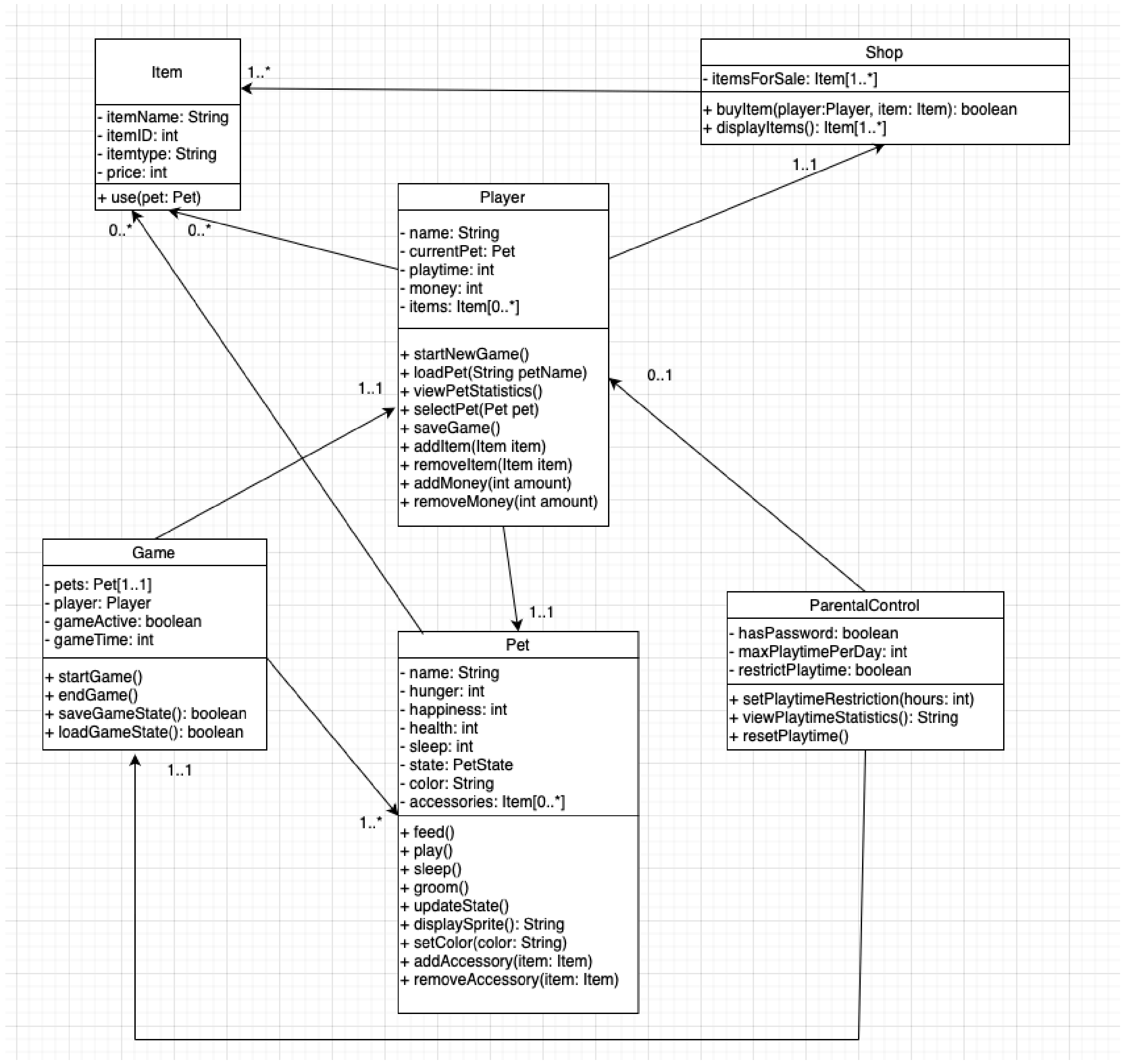
- Transform the functional and non-functional requirements into a concrete architectural design
- Establish a clear class structure and relationship model that supports all required game features
- Define an intuitive and engaging user interface that appeals to the target audience
- Document data structures and persistence mechanisms for saving game state
- Outline appropriate design patterns to ensure code quality and maintainability
- Provide a comprehensive reference for all team members during implementation
- Establish a foundation for testing strategies in subsequent project phases

## References

- CS2212 Virtual Pet Game Project Specification
- Team 81 Requirements Documentation
- UI/UX Balsamiq Source File

# Class Diagrams

Class Diagram: Class Diagram



## Overview of the Design:

This class diagram represents the core logic of our virtual pet game. The key entities include Player, Pet, Shop, and Item, each serving a distinct role in managing gameplay interactions.

## Player

- The Player class represents the user playing the game. It tracks the player’s name, playtime, in-game currency (money), and inventory (items). The player owns exactly one pet at a time, which they can interact with, feed, and customize.
- The player can purchase items from the Shop using in-game currency and store them in their inventory. Items such as food, toys, and accessories can then be used to interact with their pet. The class includes methods to manage these interactions, modify player attributes, and swap between pets if future features allow multiple pet ownership.

## Item

- Items will have 3 types in our game: food, toys, and accessories
- Items can be bought by the player from the shop for some amount of money
- Contains items for the player to buy
- Has a list of items for sale, can invoke methods to subtract currency from player on purchase

## Pet

- The Pet class represents the virtual pet that the player owns and interacts with. It has attributes such as hunger, happiness, health, and accessories, which change based on the player’s actions.
- The pet’s well-being is influenced by feeding, playing, and equipping accessories. Methods such as feed(), play(), and updateState() allow the player to care for their pet. Pets also have visual attributes such as color and accessories, which can be modified using the player's inventory.

## Shop

- The Shop allows players to purchase items such as food, toys, and accessories. It maintains a list of available items, and when a player buys an item, the Shop deducts the item's price from the player's money before adding it to their inventory.
- The Shop does not store player-owned items, meaning purchased items are only transferred to the Player's inventory. The displayItems() method allows players to see the available options before making a purchase.

## Item

- Categorized into Food, Toy, and Accessory. Used to interact with the pet.
- Can be bought from the shop with in-game currency.

## Game

- The Game class manages the entire game state, handling player progress, pet interactions, inventory updates, and saving/loading. It ensures that player actions, such as purchasing an item or feeding a pet, are correctly processed.
- The game maintains a list of all pets, the active player, and the session state. It includes functions to start and end a session, update progress, and store data persistently. The saveGameState() and loadGameState() methods ensure that player data is retained between sessions.

## ParentalControl

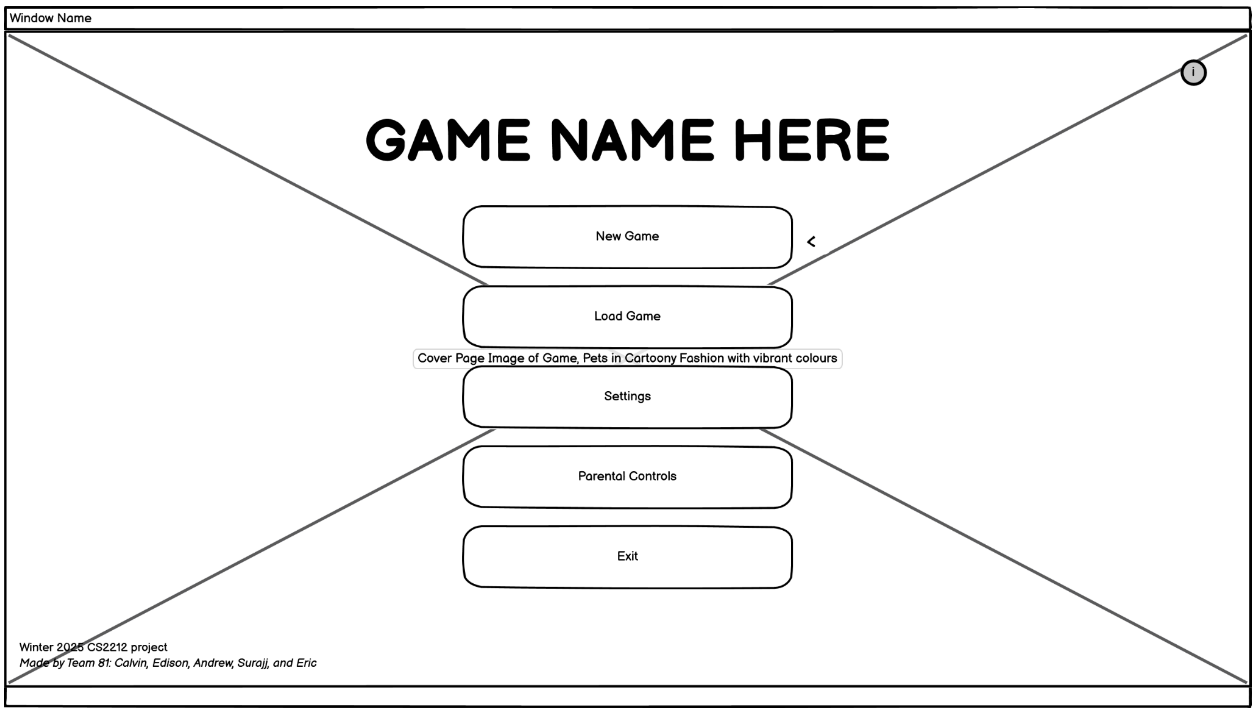
- The ParentalControl system restricts playtime for younger users by setting a maximum playtime per day. If enabled, it prevents the player from exceeding a set number of gaming hours.
- This class provides methods for setting restrictions, viewing statistics, and resetting playtime limits. The system can be toggled on or off, depending on whether a parent wants to enforce restrictions for the player.

## Relationships Between Classes

- Players have a 1..1 association with Pet, meaning each player owns exactly one pet.
- Players can own multiple items (0..*), while the Shop sells multiple items (1..*).
- ParentalControl is an optional (0..1) feature linked to the Player, restricting gameplay if enabled.

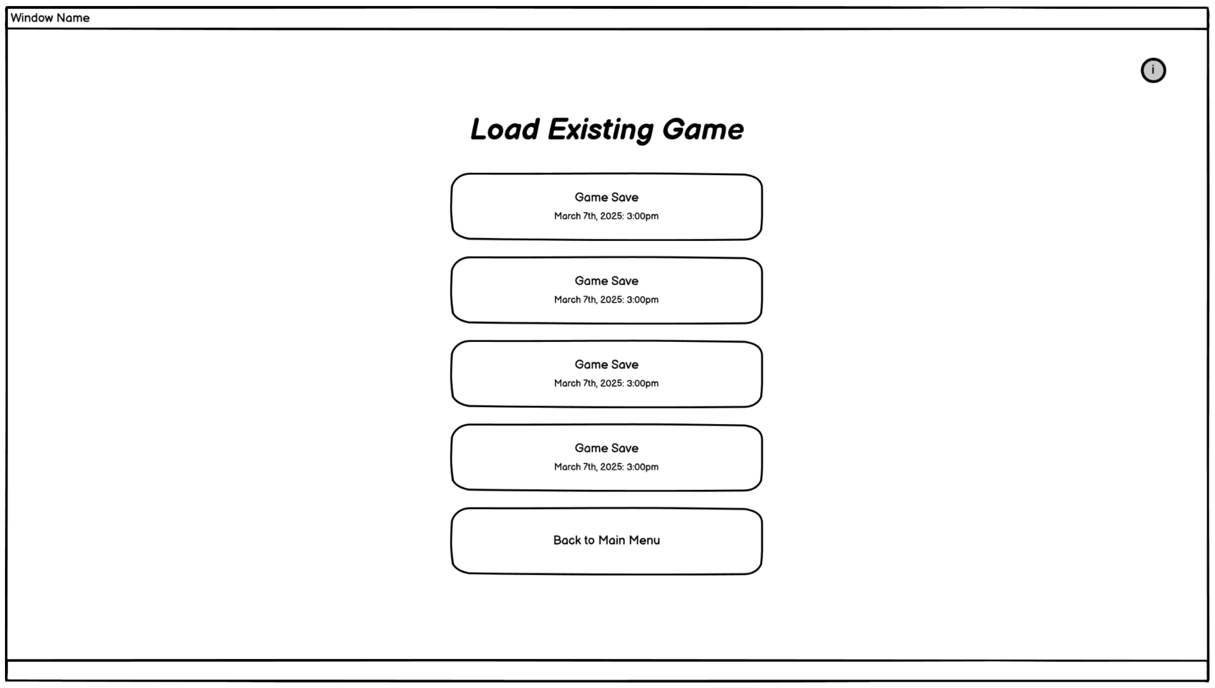
## User Interface Mockup

### Main Menu


Window Name
GAME NAME HERE
New Game
<
Load Game
Cover Page Image of Game, Pets in Cartoony Fashion with vibrant colours
Settings
Parental Controls
Exit
i
Winter 2025 CS2212 project
Made by Team 81: Calvin, Edison, Andrew, Surajj, and Eric

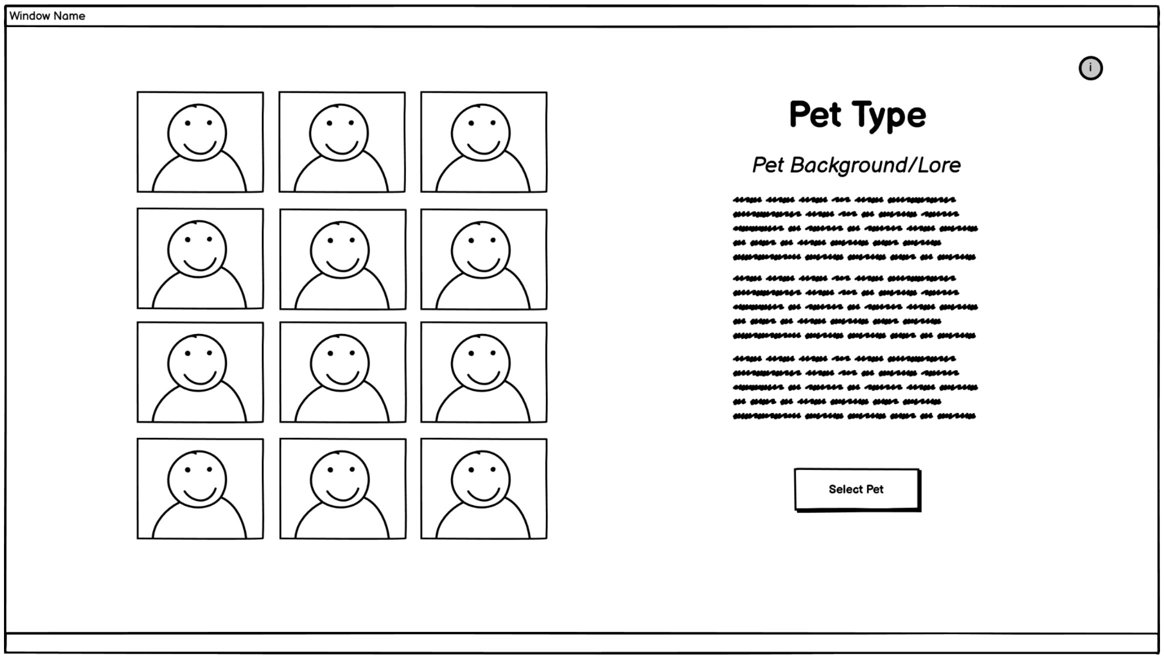
- We chose to have a simple main menu which follows basic design principles. Our rounded buttons for new/load game, settings, parental controls, and exit provide a smoother feel
- Large title text shows users a memorable symbol that they can associate the game with
- Background image of the game/drawings of the various pets gives users a glimpse of what game they're playing, and adds to the immersiveness from the first screen
- The little arrow is a quality-of-life feature that indicates which selection the user is on. Hovering over a button will trigger the button to pop up
- Info button for help

## Load Game



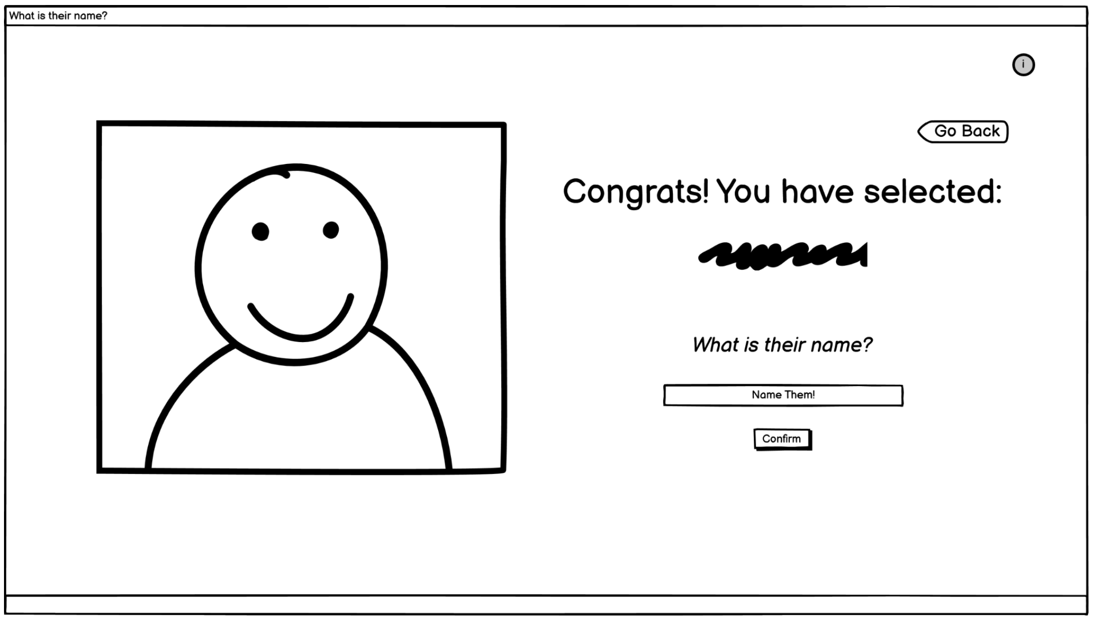
- Load games, with dates and times so that the user remembers where they left off
- Return to main menu button to return
- Info button for help

## Pet Select



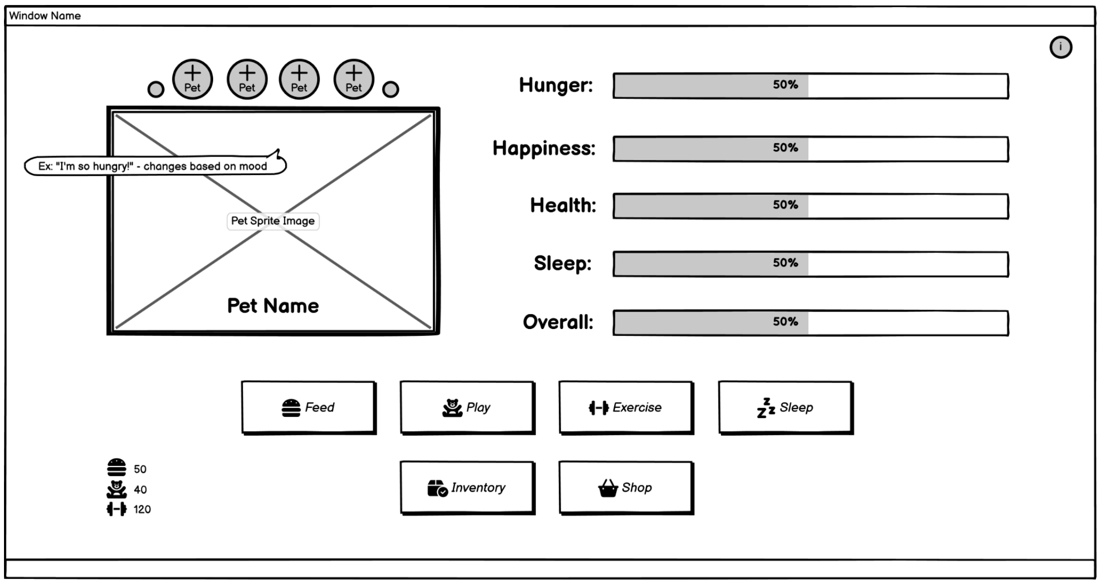
- Users can see the sprites of different pets so that they know what they like visually
- Hovering over a pet displays its history, adding immersiveness to the pets themselves through history
- Pet select button to select the pet

## Name Pet



- Shows the pet they select and give them a name with the text box
- Info button for help

## Pet Interaction

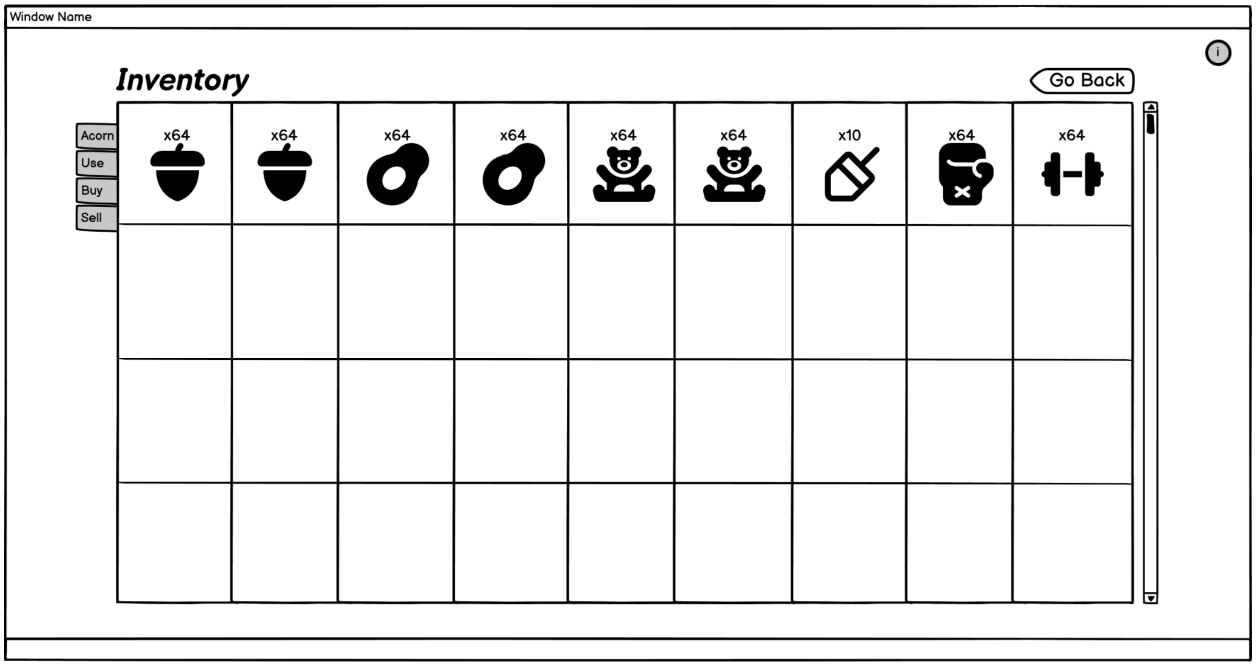


- An overall display of the pet's sprite, a text box of their mood ("I'm so hungry", "I'm sad", etc.), and action buttons that feed the pet are the general functionalities that are shown. These are core and integral to the game, and especially the text box adds immersiveness. The icons attached to the buttons make the interface consistent, and natural/metaphorical
- On the bottom left, small icons consistent with the buttons show the remaining food, weights, and toys. The user has to refer to this to know when they should buy more in the shop
- On top of the pet sprite, is a carousel of other pets that the user can switch to. We thought this was more intuitive and fun than a traditional pet selection screen, as the

user can simply click on the pet they want to switch to. Clicking on the edge buttons moves the carousel if the user has more pets that aren't shown in the initial 4

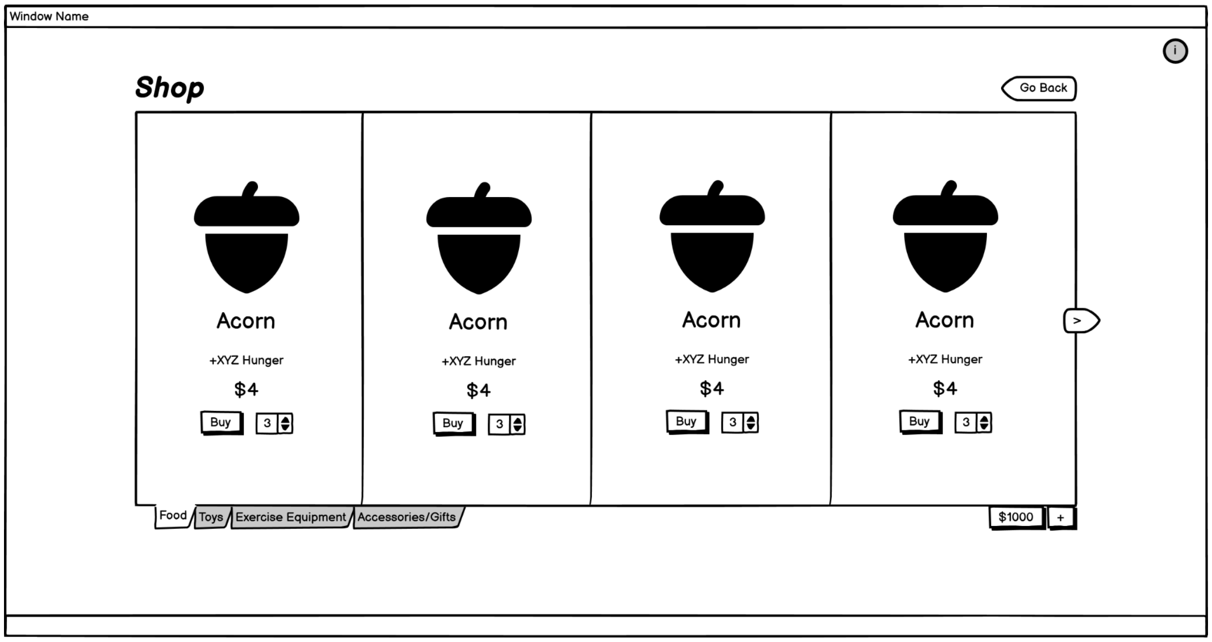
- Info button for help

## Inventory



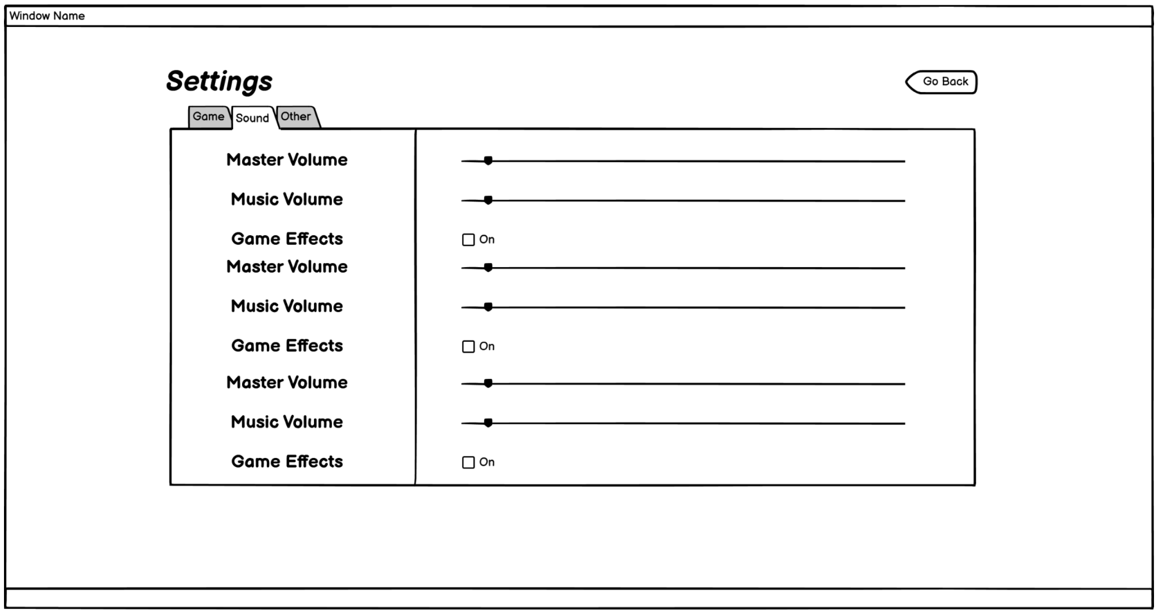
- In the inventory, we decided we wanted to showcase every item as an icon to make users understand visually what each item is
- A number above "x64" is the maximum stack of an item, because we want to make sure that users are conscious of their inventory space, and having no limit would be difficult
- Hovering over an icon will show the use, buy, sell buttons, allowing a user to perform those actions. Additionally, it shows in text what the item is, just in case the user doesn't know from the icons (which are consistent across the application)
- Back to pet button helps user just back out
- Info button for help

## Shop



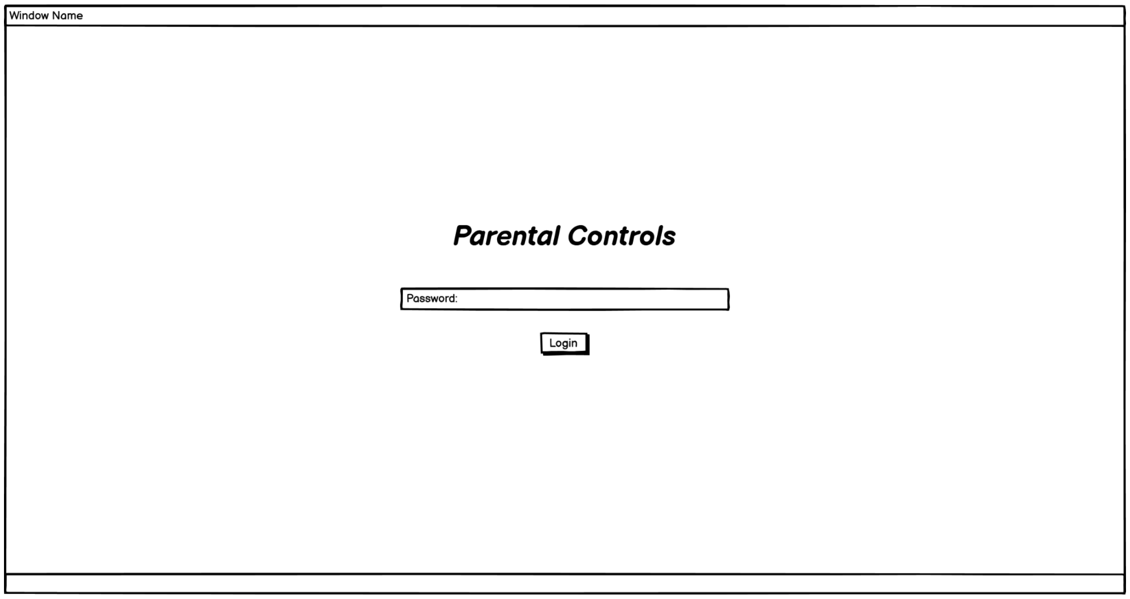
- Carousel shop format allows more detailed descriptions of the items, while maintaining fidelity and simplicity
- Tabs organize items according to their category
- Users have the option to buy many items at once to prevent repetitive clicking of the "buy" button
- Info button for help

## Settings



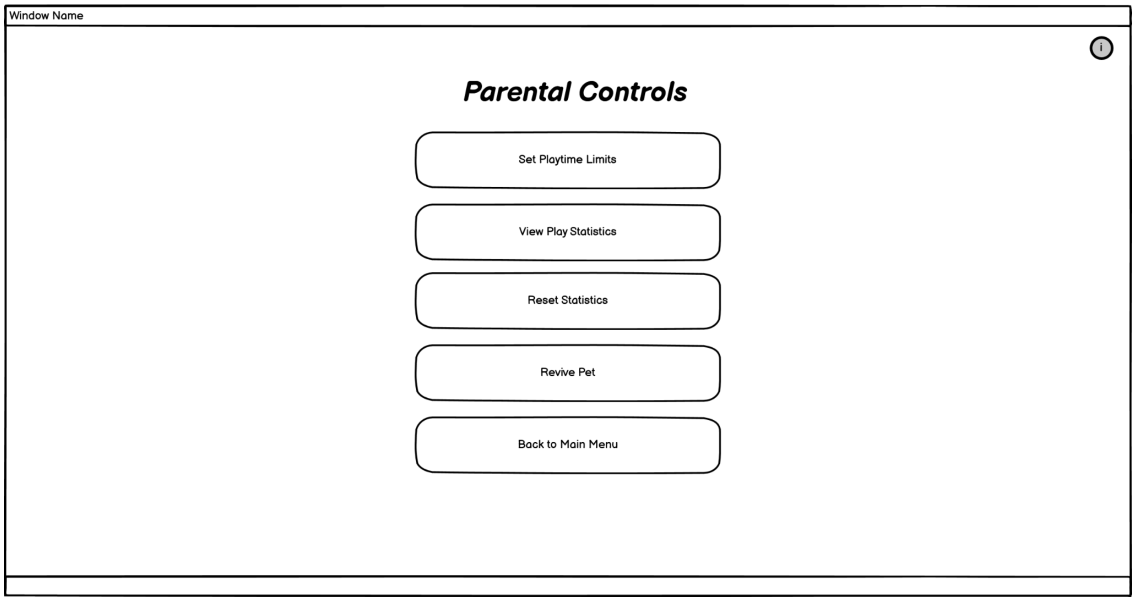
- Game, Sound, and Other tabs for different setting use cases
- Separating slider and the label allows for cleaner UI and sliders are used for items where granularity in the selection is important (having 52% sound instead of just 50%)
- Back button to go back to the screen you just came from

## Login



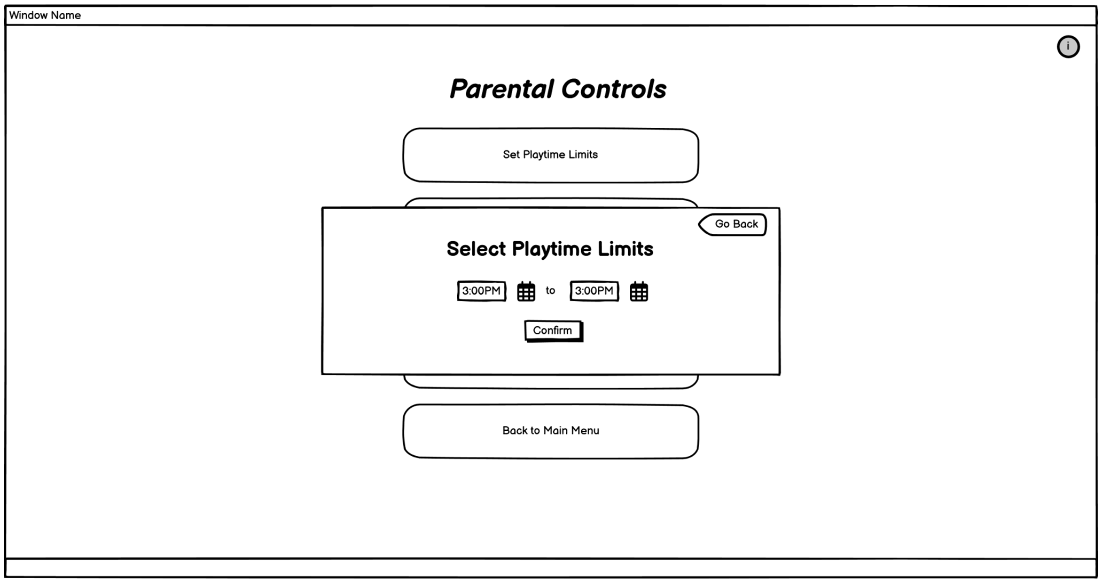
- Simple password text box to enter, with feedback and verification

## Parental Controls



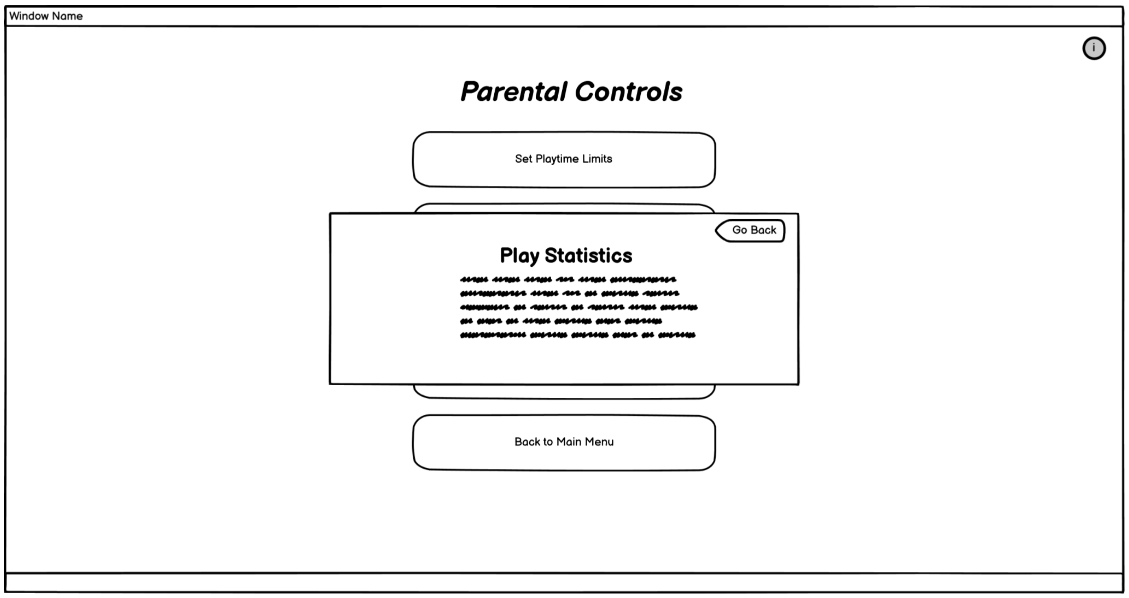
- Simple layout because parents generally are older, and have a bit more trouble relating to the traditional game design layouts
- Info button for help

## Playtime Limits



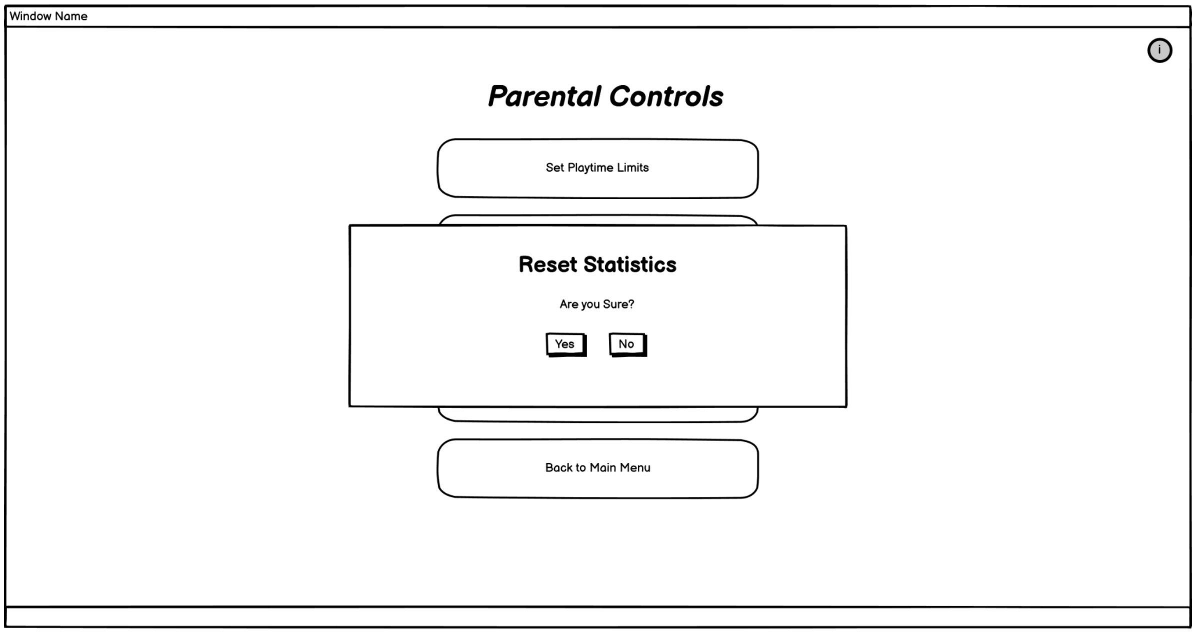
- Select times that the user can click into and choose from times every day, and different dates
- Confirm button so that the user doesn't accidentally confirm a time they don't want

## View Statistics



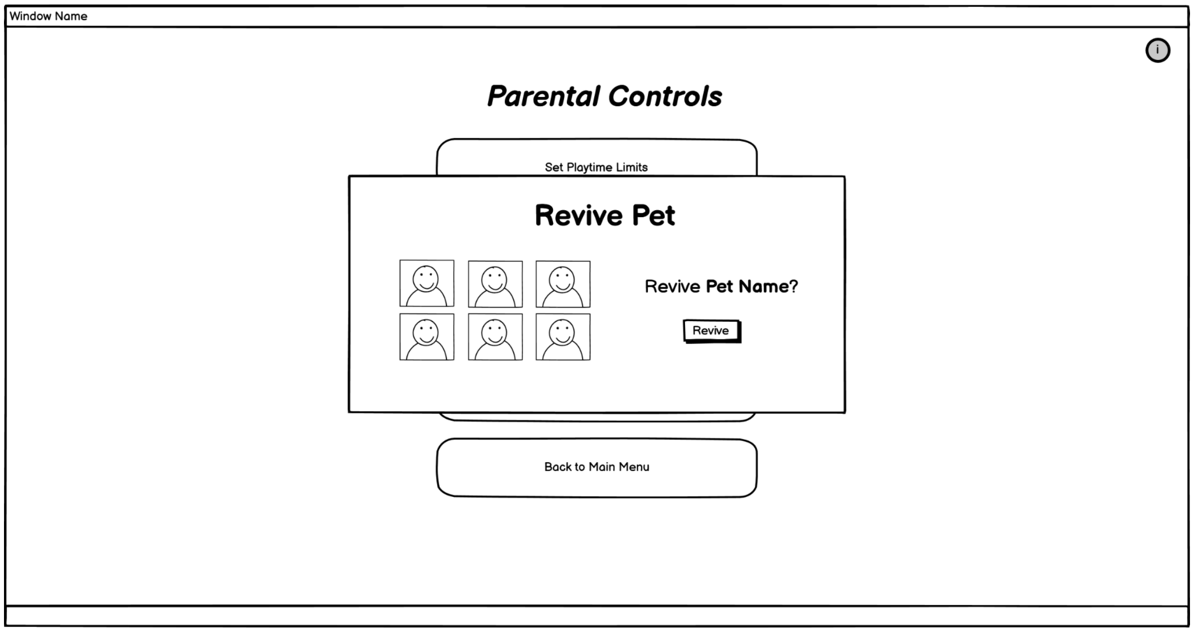
- Display the statistics in text format

## Reset Statistics



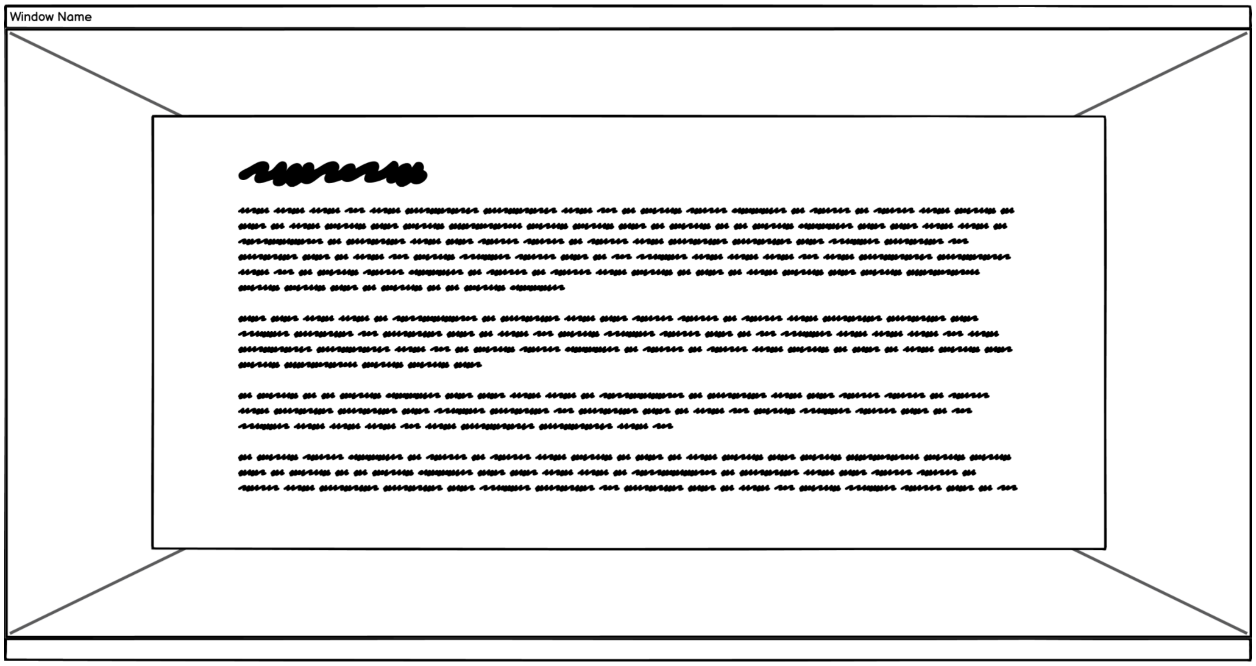
- Yes will reset, no will bring back to main menu. Having a go back button is redundant here

## Revive Pet



- Hovering over the dead pets shown and clicking them will result in the revive text showing up. Consistent with the pet selection screen

## Help



- Every screen has a small "i" circle that users can click for help, which takes them to a variation of this wireframe
- The background is the wireframe they just left, but blurred out, giving the impression they "paused" the game to get some help and information. Adds to the immersiveness
- Instructions in text are given for help

# File Formats

## Overview

For this project, we have chosen CSV (Comma-Separated Values) as the primary format to store game data. This decision was made based on several key advantages:

- Java provides robust built-in and external libraries for handling CSV files
- CSV files can be easily transferred between different systems and applications
- The format offers a good balance of simplicity and functionality for our data storage needs
- CSV files are human-readable, making debugging and manual inspection straightforward

Each CSV file will store different aspects of the game data, with a clear structure and organization as detailed below.

## External CSV Libraries

We will be using the **OpenCSV** library for CSV file handling. OpenCSV was selected because it provides:

- Robust CSV parsing with automatic handling of quotes and commas within fields
- Support for custom separators to accommodate various data formats
- Simple bean binding which maps Java objects directly to CSV records
- Well-maintained documentation and active community support
- Stability and reliability for production-level applications

## File I/O Operations

A centralized approach to file operations will be implemented through a **DataManager** class that:

- Handles all CSV file read and write operations
- Uses OpenCSV's CSVReader with header mapping to automatically convert CSV rows to Java objects
- Employs CSVWriter with bean-to-CSV conversion to serialize our objects back to CSV format
- Opens files in try-with-resources blocks to ensure proper closing even when exceptions occur
- Implements caching mechanisms to improve performance for frequently accessed data

## Data Validation

To maintain data integrity, we will implement comprehensive validation strategies:

- All fields will be validated before writing to CSV (e.g., checking for valid email formats, appropriate ranges for numeric values)
- Required fields will be verified for null or empty values to prevent data corruption
- Custom validators will handle complex validations like date formats and logical constraints
- Invalid data will trigger user-friendly error messages rather than system crashes

- Data repair mechanisms will attempt to recover from common corruption issues

## CSV File Structures

### players.csv

**Purpose**: Stores information about players.

| Column | Data Type | Description |
|---|---|---|
| player_id | String | Unique identifier for each player |
| username | String | Player's in-game name |
| email | String | Email address of the player |
| join_date | Date | Date when the player joined |
| last_login | Timestamp | Timestamp of the last login |
| currency | Integer | Amount of in-game currency |

### virtual_pets.csv

**Purpose**: Stores details of the player's virtual pets.

| Column | Data Type | Description |
|---|---|---|
| pet_id | String | Unique identifier for each pet |
| player_id | String | Links pet to the respective player |
| pet_name | String | Name of the pet |
| species | String | Type of pet (e.g., dog, cat, dragon) |
| age | Integer | Pet's age in days |
| hunger_level | Integer | Numeric representation (0-100) |

| | | |
|---|---|---|
| happiness_level | Integer | Numeric representation (0-100) |
| health_status | Integer | Health level of the pet |
| last_fed | Timestamp | Timestamp of the last feeding |

### game_progress.csv

**Purpose**: Stores game progress for each player.

| Column | Data Type | Description |
|---|---|---|
| player_id | String | Foreign key linking to players.csv |
| level | Integer | Current level of the player |
| experience_points | Integer | Total XP earned |
| quests_completed | Integer | Number of completed quests |
| badges_earned | String | Comma-separated list of badges |

### transactions.csv

**Purpose**: Tracks in-game purchases and currency transactions.

| Column | Data Type | Description |
|---|---|---|
| transaction_id | String | Unique identifier for each transaction |

| | | |
|---|---|---|
| player_id | String | Foreign key linking to players.csv |
| transaction_type | String | Type of transaction (e.g., "purchase", "reward", "penalty") |
| amount | Integer | Amount of in-game currency involved |
| timestamp | Timestamp | Time of transaction |

## inventory.csv

**Purpose**: Stores the items players own.

| Column | Data Type | Description |
|---|---|---|
| player_id | String | Foreign key linking to players.csv |
| item_id | String | Unique identifier for the item |
| item_name | String | Name of the item |
| quantity | Integer | Number of this item owned |
| type | String | Category of item (Food, gift, or other) |

## parental_controls.csv

**Purpose**: Stores parental restrictions and statistics.

| Column | Data Type | Description |
|---|---|---|
| player_id | String | Foreign key linking to players.csv |
| time_limit_enabled | Boolean | Whether playtime restriction is active |
| allowed_playtime_range | String | Permitted play hours (e.g., "14:00-18:00") |
| total_playtime | Float | Total playtime in hours |
| average_session_length | Float | Average length of a play session |

# Development Environment

## IDE

The IDE our team will be using is IntelliJ

## External Libraries

- JUnit 5 for unit testing (required)
- FasterXML/jackson to handle JSON serialization/deserialization

## Version Control

- Gitlab will be used for version control, with branching to allow for seamless collaboration
- Regular commits will be used throughout development, with descriptive commit messages

# Patterns

## 1. Input Feedback

### Overview

Input feedback is a crucial aspect of user interaction design that ensures users receive immediate and clear responses from the system when they perform actions. This can enhance usability by making users feel informed and confident in their interactions.

### Why it is Appropriate for the Project

In a virtual pet game, feedback is essential for guiding both players and parents in their actions. Without feedback, users might feel lost or unsure whether their actions were registered.

### Application in the Project

- **Parental Controls**: Provide visual or auditory feedback when settings are changed, such as displaying confirmation messages when enabling/disabling restrictions.
- **Player Commands**: When the player issues a command to the pet (e.g., feeding, playing), feedback should be provided through:
  - Animations of the pet reacting.
  - Sound effects confirming the action.
  - Changing button states (e.g., disabling buttons for commands that are on cooldown).
- **Inline Alerts and Notifications**: Error messages for invalid actions (e.g., trying to feed the pet when inventory is empty) should be clear and informative.

---

## 2. Settings

### Overview

The settings pattern provides a structured way to manage user preferences by ensuring they are accessible, well-organized, and easy to navigate.

### Why it is Appropriate for the Project

A well-structured settings menu enhances usability and ensures that users can efficiently manage their preferences without frustration.

### Application in the Project

- **Clustering Preferences**:
  - A small number of settings (≤7) should be displayed as a simple list.
  - Larger sets (8–16) should be grouped under labeled sections.
  - If settings exceed 16, submenus should be introduced for better organization.
    - These are just guidelines. We can slightly bend the rules if appropriate
- **Parental Controls Menu**: Options for setting playtime restrictions, viewing statistics, and reviving pets should be categorized separately.
- **Default Settings**: Sensible defaults should be chosen to minimize the need for user adjustments while preventing unintended risks.

---

## 3. Variable Rewards

### Overview

The variable rewards pattern leverages randomness to create engagement and motivation. Users are more likely to stay engaged when rewards are unpredictable but fair.

### Why it is Appropriate for the Project

A sense of unpredictability keeps players engaged and encourages them to interact regularly with their pet.

### Application in the Project

- **Pet Reactions**: The pet should occasionally exhibit different responses to the same command, maintaining novelty.
- **Item Acquisition**: When obtaining items (e.g., food, gifts), a randomized system could determine the specific item received, ensuring variety.
- **Avoiding Extinction**: Players should always receive some form of reward or acknowledgment, even if minimal, to prevent disengagement.
- **Negative Rewards**: If the pet is neglected, it should exhibit displeasure, but not to the extent that discourages the player entirely.

---

## 4. Praise

### Overview

Encouraging positive behavior through praise can reinforce desired actions, making them more likely to be repeated.

### Why it is Appropriate for the Project

A virtual pet game should reward players for taking care of their pet, reinforcing the habit of regular interaction.

### Application in the Project

- **Visual and Audio Feedback**: When the pet is in a good state, it should display a happy animation and play a cheerful sound effect.
- **Score Rewards**: Taking good care of the pet should increase the player's score.
- **Notifications**: Text-based praise messages (e.g., "Your pet loves playing with you!") can further reinforce positive actions.

---

## 5. State Pattern

### Overview

The state pattern is used to manage an object's behavior based on its current state. It involves creating separate classes for different states, with a context class handling transitions.

### Why it is Appropriate for the Project

The pet in this game has clearly defined states (e.g., happy, hungry, angry, asleep, dead). Managing these states using the state pattern ensures better modularity and easier maintenance.

### Application in the Project

- **Separate State Classes**: Each pet state (HappyState, HungryState, AngryState, SleepingState, DeadState) should be implemented as a class.
- **Context Class**: The pet class should delegate behavior to the current state object, allowing dynamic transitions.
- **State Transitions**: Defined transitions between states ensure consistency in gameplay logic.

---

## 6. Command Pattern

### Overview

The command pattern encapsulates actions as objects, allowing requests to be parameterized, queued, and executed dynamically.

### Why it is Appropriate for the Project

In a virtual pet game, player actions (e.g., feeding, playing, exercising) can be encapsulated as commands, making it easier to manage undo/redo functionality and implement cooldowns.

### Application in the Project

- **Encapsulation of Player Actions**:
  - `FeedCommand`: Increases fullness and decreases item count.
  - `PlayCommand`: Increases happiness and triggers a cooldown.
  - `ExerciseCommand`: Affects multiple stats (hunger, health, sleep).
- **Invoker Component**: The UI should send commands to an invoker that manages their execution.
- **Undo Feature (Optional)**: Parents might have an option to undo recent actions in case of mistakes.

---

## Sources:

https://ui-patterns.com/patterns/InputFeedback
https://ui-patterns.com/patterns/settings
https://ui-patterns.com/patterns/Variable-rewards
https://ui-patterns.com/patterns/Praise
https://refactoring.guru/design-patterns/state
https://www.oodesign.com/command-pattern

# Summary

This design documentation has outlined the comprehensive approach our team is taking to develop the Virtual Pet Game. By leveraging object-oriented design principles, we've created a modular class structure that separates game logic from user interface components while maintaining clear relationships between entities such as Player, Pet, and Shop. Our user interface design prioritizes intuitiveness and engagement through consistent navigation patterns, visual feedback, and age-appropriate aesthetics. The data management strategy using CSV files with the OpenCSV library provides a straightforward yet robust solution for persisting game state across sessions. The development environment choices of IntelliJ IDE, JUnit testing framework, and version control through GitLab support collaborative and quality-focused development practices. The implementation of established design patterns will ensure the codebase remains maintainable and extensible as the project evolves. Together, these design elements form a solid foundation for implementing an engaging, educational, and family-friendly virtual pet experience that meets the project requirements while allowing for future enhancements.

# Project Management

All design documentation tasks were tracked using GitLab issues with appropriate labels to indicate priority, difficulty, and category (e.g., "UI Design," "Class Diagram," "File Format"). This approach provided full transparency into task status and accountability. Progress updates and comments were added directly to relevant issues, creating a historical record of design decisions and revisions.

The issue board can be accessed at: Git Lab Issues

## Task Assignment and Tracking

Tasks were assigned during our initial planning meeting for this milestone, with consideration for each member's strengths, interests, and workload:

1. **Initial Assignment**: Tasks were distributed based on the contributors table shown on the main page, with clear deadlines for each component.

2. **Progress Tracking**: We implemented a simple 3-state tracking system:

    - Todo: Task assigned but not yet started
    - In Progress: Work actively underway
    - Review: Ready for team review before marking as complete
    - Complete: Finished and approved by team
3. **Workload Balance**: We monitored task completion rates and adjusted assignments as needed to ensure balanced contributions. When Calvin completed the UI mockups

ahead of schedule, he assisted Andrew with documenting class relationships.

4. **Blockers and Dependencies**: We identified critical path tasks early and prioritized them to prevent bottlenecks. For example, the class diagram needed to be at least partially complete before file formats could be finalized.

5. **Deadline Management**: Each team member committed to specific internal deadlines (typically 2-3 days before final submission) to allow time for integration and review.

## Team Meetings and Collaboration

Our team conducted regular meetings throughout the development of this milestone:

All meeting minutes were documented and stored in our GitLab repository for reference. The minutes captured attendance, discussion points, decisions made, action items, and follow-up responsibilities. These records ensured accountability and provided a clear history of our design evolution. This also includes our TA meetings.

Meeting minutes can be accessed at: Team Meeting Minutes

Our collaborative approach extended beyond formal meetings through:

- Regular asynchronous communication via Discord
- Pair programming sessions for complex design components
- Structured peer review process for all documentation sections
- Clear documentation of design decisions and rationales

## Glossary of Terms

| Term | Definition |
|---|---|

| | |
|---|---|
| API | Application Programming Interface - a set of rules that allows different software programs to communicate with each other. |
| Attribute | A piece of data that belongs to a class, like a pet's name or hunger level. |
| Class | A blueprint for creating objects in programming that defines what properties and behaviors the objects will have. |
| Class Diagram | A visual representation of the classes in the system and how they relate to each other. |
| CSV | Comma-Separated Values - a simple file format that stores data in a table where each row is on a new line and columns are separated by commas. |
| Design Pattern | A reusable solution to a common problem in software design. |
| IDE | Integrated Development Environment - a software application that provides tools for programming, like code editing and testing. |
| Interface | A way for the player to interact with the game, including screens, buttons, and visual elements. |
| JUnit | A testing framework for Java programs that helps ensure the code works correctly. |
| Method | An action that an object can perform, like feeding a pet or buying an item. |
| Mockup | A visual draft of how the user interface will look before it's actually built. |
| Object | A specific instance of a class in programming, like a particular pet in the game. |
| Parental Controls | Features that allow parents to monitor and limit a child's playtime in the game. |
| UML | Unified Modeling Language - a standard way to visualize the design of a system using diagrams. |
| UI | User Interface - what the player sees and interacts with when playing the game. |
| Version Control | A system that records changes to files over time so specific versions can be recalled later. |
| Wireframe | A basic outline of a user interface showing layout and functionality without detailed design elements. |