CS 5004 Final Synthesis

Introduction:

This application is a virtual pet simulation game that allows users to adopt, interact with, and manage various types of pets, such as dragons, unicorns, and robots. Users can perform actions like feeding, playing, and grooming their pets to maintain their health and mood. The game includes features like displaying available pets, adopting pets, checking the user's balance, and interacting with adopted pets. When a user interacts with their adopted pet, their mood + health will change accordingly. Similarly, when a user adopts a pet, their balance will reflect that by subtracting the price of the animal from that.

Here’s a break down of the code:

The main application, called "VirtualPetGame", acts as the entry point and orchestrates the overall game flow. It initializes the game controller, view, and other necessary components.

The GameController class manages the game logic, including adopting pets, interacting with them, tracking the user's balance, and maintaining the list of available and adopted pets.

To represent different types of pets (Dragons, Unicorns, and Robots), I created separate classes that inherit from an abstract base class called "AbstractPet". This base class defines common attributes and behaviors shared by all pet types, such as name, price, health, mood, and adoption status. The concrete pet classes (Dragon, Unicorn, and Robot) override the abstract methods to implement type-specific behaviors for feeding, playing, and grooming.

The GameView class handles the user interface and interaction. It displays the main menu, prompts the user for input, and facilitates the adoption and interaction processes with pets.

The LinkedList class is a custom implementation of a linked list data structure used to store and manage the list of available pets.

To support the virtual pet game concept, I created the following smaller applications or components:

- Pet Interface: This interface defines the common behaviors that all pet types should implement, such as getting the pet's name, price, health, mood, adoption status, and type, as well as methods for feeding, playing, and grooming the pet.
- Node Class: This class represents a node in the linked list data structure, containing the pet data and a reference to the next node.
- LinkedList Class: This class implements the linked list data structure, providing methods to add, retrieve, and manipulate pets in the list.

One notable aspect of my implementation is the use of recursion in the GameController class for counting available pets of a specific type and counting adopted pets.

Additionally, I utilized Java's stream API and lambda expressions to perform operations on the list of pets, such as filtering, mapping, and collecting pets based on specific criteria (e.g., getting adopted pets, getting pets by type).

The game flow works as follows:

The user is presented with a main menu where they can choose to adopt a pet, interact with their adopted pets, display all pets, check their balance, or exit the game.

If the user chooses to adopt a pet, they are prompted to select the type of pet they want to adopt (Dragon, Unicorn, or Robot). The available pets of the selected type are displayed, and the user can choose which pet to adopt if they have enough balance.

If the user chooses to interact with their adopted pets, they are shown a list of their adopted pets and can select one to interact with. They can then choose to feed, play, or groom the selected pet. When they choose to interact with their pet, the pet's mood and health will be affected accordingly.

The user can also display all available and adopted pets, as well as check their current balance.
The game continues until the user chooses to exit.

CONCEPT MAP:

Concept	Location	How it's implemented
Recursion	GameController.Java 1. countAvailablePetsRecursive 2. countAdoptedPetsRecursive	Two recursive methods, countAvailablePetsRecursive and countAdoptedPetsRecursive, are employed to iterate through the list of pets and count them based on their adoption status and type. countAvailablePetsRecursive: This method recursively traverses the list of remaining pets, checking each pet's adoption status and type. If the pet is not adopted and matches the specified type, it increments the count of available pets. This process continues until there are no more pets left to process. countAdoptedPetsRecursive: Similarly, this method recursively traverses the list of remaining pets, incrementing the count of adopted pets if the current pet is already adopted. It continues this process until there are no more pets left to process. When the player interacts with the application, the output displays the number of adopted and available pets based on the recursive counting

		methods. For example, the output may show "Number of Adopted Pets: 3" and "Number of Available Pets: 7", providing the player with a clear overview of the current pet population in the game.
Logical Structure and Design using Abstract Classes and Interfaces	AbstractPet.java, Dragon.java, Unicorn.java, Robot.java	Abstract Class: The AbstractPet class serves as the foundation for all types of pets in the application. It encapsulates common attributes such as name, price, health, mood, and adoption status. Additionally, it implements the Pet interface, providing method signatures for actions like feeding, playing, and grooming. Interfaces: The Pet interface defines the contract for all pet types, ensuring that each pet class implements essential methods like getName(), getPrice(), feed(), play(), and groom(). This ensures consistency in behavior across different pet types while allowing for polymorphic behavior. Concrete Classes: Concrete pet classes like Dragon, Unicorn, and Robot extend the AbstractPet class and provide specific implementations for their respective behaviors. They override methods like feed(), play(), and groom() to reflect the unique characteristics of each pet type.

Useful and Logical Abstraction using Generics and Lambda Expressions	LinkedList.java
	GameController
		class LinkedList<T> {
		    Node<T> head;
		    // rest of code
		}
		
		The LinkedList<T> class is implemented using generics, allowing it to store elements of any data type. This enables the creation of a flexible and reusable linked list data structure that can be used to store various types of objects, including pets.
		
		The Lambda expressions are used with the stream() API in the ‘GameController’ class. These are some of the examples:
		
		//Example 1: Filtering pets based on specific criteria using a lambda expression
		
		public boolean canAdoptPet() {
		    int totalPrice = pets.stream()
		                        .filter(pet -> !pet.isAdopted())
		                        .map(Pet::getPrice)
		                        .mapToInt(Integer::intValue)
		                        .sum();
		    return totalPrice <= balance;

		}
		
		// Example 2: Mapping attributes of pets to integers for calculations using lambda expressions
		
		public LinkedList<Pet> getAdoptedPets() {
		return pets.stream()
		.filter(Pet::isAdopted)
		
		.collect(Collectors.toCollection(LinkedList::new));
		}
		
		When the player interacts with the pet simulation, lambda expressions are used behind the scenes to filter, map, and aggregate pet data based on user actions.

Higher Order Functions Map, Filter, and Fold	GameController.java	In the GameController class, the canAdoptPet() method uses filter and map to calculate the total price of available pets (not adopted):
		
		int totalPrice = pets.stream()
		.filter(pet -> !pet.isAdopted())
		.map(Pet::getPrice)
		.mapToInt(Integer::intValue)
		.sum();
		
		Here, filter is used to get only the pets that are not adopted, and map is used to extract the price of each pet from the filtered list. This allows the method to determine if the user has enough balance to adopt a pet.
		
		1. In the GameController class, the getPetsByType(String type) method uses filter to get a list of pets of a specific type:
		
		public LinkedList<AbstractPet> getPetsByType(String type) {
		return pets.stream()
		.filter(pet -> pet.getType().equals(type))
		
		.collect(Collectors.toCollection(LinkedList::new));
		}
		
		Here, filter is used to include only the pets whose type matches the provided type parameter. This is how it in the output, we see all the dragons, unicorns, and robots separately.

		2. In the GameController class, the getAdoptedPets() method uses filter to get a list of adopted pets:
		
		return pets.stream()
		        .filter(Pet::isAdopted)
		
		.collect(Collectors.toCollection(LinkedList::new));
		
		Here, filter is used to include only the pets that have been adopted.
		
		3. In the GameController class, the getAvailablePets() method uses filter to get a list of available (not adopted) pets:
		
		return pets.stream()
		        .filter(pet -> !pet.isAdopted())
		
		.collect(Collectors.toCollection(LinkedList::new));
		
		Here, filter is used to include only the pets that have not been adopted.

Hierarchical Data Representation as an ADT or a Linked List ADT	LinkedList.java Node.Java	In my code, I am using an implementation of a linked list data structure to represent the list of available pets. This is evident in the LinkedList and Node classes I have created. The LinkedList class represents the hierarchical data structure itself, where each node in the list can have a reference to the next node, forming a linear chain or hierarchy. The Node class represents an individual node in the linked list, containing the pet data (data field) and a reference to the next node (next field). Here are the relevant parts of my code that demonstrate the use of the linked list data structure: The LinkedList class: **class LinkedList<T> {** **Node<T> head;** **// Methods to add, get, check size, etc.** **// ...** **}** This class encapsulates the linked list data structure and provides methods to manipulate the list, such as adding nodes, getting nodes at a specific index, and checking the size of the list.

		The Node class:
		
		class Node<T> {
		T data;
		Node<T> next;
		
		public Node(T data) {
		this.data = data;
		this.next = null;
		}
		}
		
		This class represents an individual node in the linked list, containing the pet data (data field) and a reference to the next node (next field).
		
		Usage of the LinkedList in the GameController class:
		
		private LinkedList<Pet> pets;
		
		public GameController() {
		this.pets = new LinkedList<>();
		// Add pets to the list
		pets.add(new Dragon("Fireball", 50, 90, "Calm"));
		pets.add(new Unicorn("Sparkles", 75, 80, "Happy"));
		// ...
		}
		
		In the GameController class, I created an instance of the LinkedList class to store the list of available pets. I then added pet objects (instances of Dragon, Unicorn, and Robot classes) to this linked list using the add method. The linked list data structure allows me to efficiently manage the list of pets, bc I can easily add, remove, or access pets at any position in the list without the need for shifting elements, compared to something like an array-based implementation.
		
		Throughout my code, I use the LinkedList instance to perform various operations on the list of pets, such as adopting pets, counting available pets, filtering pets by type or adoption status, and displaying the list of available or adopted pets.
		
		The linked list ADT provides a way to organize and manage the pet data in a structured manner.

Architectures	Pet.Java	In my code, I have followed the Model-View-Controller

and Design Patterns MVC Design and a design pattern	GameView.java GameController.java	(MVC) architectural pattern. MVC Architecture: The MVC architecture is evident in the way I have organized your classes and their responsibilities: Model: The Pet interface and the concrete pet classes (Dragon, Unicorn, and Robot) represent the model in my application. These classes encapsulate the data and behavior related to different types of pets, such as their name, price, health, mood, and adoption status, as well as the actions they can perform (feed, play, groom). View: The GameView class acts as the view component in the MVC architecture. It is responsible for displaying the user interface, prompting the user for input, and presenting the game information to the user. This class does not contain any game logic or data manipulation code. Controller: The GameController class serves as the controller in the MVC architecture. It manages the game logic, handles user interactions, and coordinates between the model (pet objects) and the view (GameView). The GameController class contains methods for adopting pets, interacting with pets, managing the user's balance, and retrieving pet information based on various criteria. By separating the concerns into these three components, I have achieved a modular and maintainable design. The model classes handle the data and behavior related to pets, the view class handles the user interface, and the controller class acts as an intermediary between the model and view, managing the application logic.
SOLID Design Principles		My code demonstrates adherence to the SOLID design principles, particularly the Single Responsibility Principle (SRP) and the Open-Closed Principle (OCP). Single Responsibility Principle (SRP): The SRP states that a class should have only one reason to change, or in other words, it should have a single responsibility. My code follows this principle by separating concerns into different classes: **The AbstractPet class is responsible for defining the common attributes and behaviors of pets.** **The concrete pet classes (Dragon, Unicorn, and Robot) are responsible for implementing the specific behaviors of each pet type.**

		The GameController class is responsible for managing the game logic and the list of pets. **The GameView class is responsible for handling the user interface and user interactions.** By separating these responsibilities into different classes, I have adhered to the SRP. Open-Closed Principle (OCP): The OCP states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. My code follows this principle by using inheritance and polymorphism to define the behavior of different pet types. **The Pet interface defines the common behaviors that all pet types should implement.** **The concrete pet classes (Dragon, Unicorn, and Robot) extend the AbstractPet class and implement the Pet interface, providing their own specific implementations of the feed(), play(), and groom() methods.**