OOP Problems

```
Prepared by Ahammad Nafiz
```

> "In the vast landscape of coding, every problem is a portal to mastery. Embrace challenges with a fearless heart, for with each line of code, you sculpt the future of technology. The elegance of your solutions is not just in syntax; it's a testament to your journey—where persistence meets innovation." - Ahammad Nafiz

Problem 1: Library Management System

Design a library management system with the following classes:

Classes:

1. **Book:**
   - Attributes:
     - `title`: String representing the title of the book.
     - `author`: String representing the author of the book.
     - `ISBN`: String representing the International Standard Book Number.
     - `available_copies`: Integer representing the number of available copies.
   - Methods:
     - `display_info()`: Display information about the book.
2. **Person:**

- Attributes:
  - `name`: String representing the name of the person.
  - `age`: Integer representing the age of the person.
- Methods:
  - `display_info()`: Display information about the person.

3. **Student (Subclass of Person):**
   - Attributes:
     - `student_id`: Integer representing the student's unique identifier.
     - `books_borrowed`: List of instances of the `Book` class that the student has borrowed.
   - Methods:
     - `borrow_book(book)`: Borrow a book and update the available copies.
     - `return_book(book)`: Return a book and update the available copies.
4. **Librarian (Subclass of Person):**
   - Attributes:
     - `employee_id`: Integer representing the librarian's employee ID.
   - Methods:
     - `add_book(book)`: Add a new book to the library.
     - `remove_book(book)`: Remove a book from the library.

Inheritance:

- **Book Inheritance:**
  - Create subclasses for different types of books (e.g., `FictionBook`, `NonFictionBook`). Each subclass should inherit from the `Book` class and may have additional attributes specific to the book type.

Encapsulation:

- **Book Class Encapsulation:**
  - Protect the `ISBN` attribute by making it private and provide a getter method to access it.

- **Librarian Class Encapsulation:**
  - Protect sensitive librarian information (e.g., employee ID) by using private attributes and providing getter methods.

Polymorphism:

- **Polymorphic Methods:**
  - Implement polymorphic methods like `display_info()` in both the `Book` and `Person` classes. This method can be used to display information about a book or a person.

Use Cases:

1. **Add and Remove Books:**
   - Create instances of the `Book` class and use the `Librarian` class to add and remove books from the library.
2. **Borrow and Return Books:**
   - Create instances of the `Student` class and use the `borrow_book()` and `return_book()` methods to manage borrowed books.
3. **Display Information:**
   - Use polymorphic methods to display information about books and people in the library.

Problem 2: Online Shopping System

Design an online shopping system with the following classes:

Classes:

1. **Product:**
   - Attributes:
     - `product_id`: Integer representing the unique identifier for the product.
     - `name`: String representing the name of the product.
     - `price`: Float representing the price of the product.
   - Methods:

- `display_info()`: Display information about the product.

2. **Customer:**
   - Attributes:
     - `customer_id`: Integer representing the unique identifier for the customer.
     - `name`: String representing the name of the customer.
     - `cart`: List of instances of the `Product` class representing the items in the customer's shopping cart.
   - Methods:
     - `add_to_cart(product)`: Add a product to the customer's shopping cart.
     - `remove_from_cart(product)`: Remove a product from the customer's shopping cart.
     - `checkout()`: Complete the purchase and display the total cost.
3. **DiscountedProduct (Subclass of Product):**
   - Attributes:
     - `discount_percent`: Float representing the discount percentage for the product.

Inheritance:

- **Product Inheritance:**
  - Create subclasses for different types of products (e.g., `ElectronicsProduct`, `ClothingProduct`). Each subclass should inherit from the `Product` class and may have additional attributes specific to the product type.

Encapsulation:

- **Product Class Encapsulation:**
  - Protect sensitive product information (e.g., price) by using private attributes and providing getter methods.
- **Customer Class Encapsulation:**

- Protect sensitive customer information (e.g., customer ID) by using private attributes and providing getter methods.

Polymorphism:

- **Polymorphic Methods:**
  - Implement polymorphic methods like `display_info()` in both the `Product` and `Customer` classes. This method can be used to display information about a product or a customer.

Use Cases:

1. **Add and Remove Products:**
   - Create instances of the `Product` class and use the `Customer` class to add and remove products from the shopping cart.
2. **Checkout and Display Total:**
   - Use the `checkout()` method to complete the purchase and display the total cost, considering any discounts for discounted products.
3. **Display Product Information:**
   - Use polymorphic methods to display information about products and customers in the online shopping system.

Problem 3: Social Media Platform

Design a simplified social media platform with the following classes:

Classes:

1. **User:**
   - Attributes:
     - `user_id`: Integer representing the unique identifier for the user.
     - `username`: String representing the username of the user.
     - `posts`: List of strings representing the posts made by the user.
   - Methods:
     - `make_post(post_content)`: Add a new post to the user's list of posts.

- `display_posts()`: Display all posts made by the user.

2. **AdminUser (Subclass of User):**
   - Attributes:
     - `admin_level`: Integer representing the administrative level of the admin user.
   - Methods:
     - `remove_post(post_content)`: Remove a post made by any user.
3. **Comment:**
   - Attributes:
     - `comment_id`: Integer representing the unique identifier for the comment.
     - `user`: Instance of the `User` class representing the user who made the comment.
     - `post_content`: String representing the content of the post being commented on.
     - `comment_content`: String representing the content of the comment.

Inheritance:

- **User Inheritance:**
  - Create subclasses for different types of users (e.g., `RegularUser`, `PremiumUser`). Each subclass should inherit from the `User` class and may have additional attributes specific to the user type.

Encapsulation:

- **User Class Encapsulation:**
  - Protect sensitive user information (e.g., user ID) by using private attributes and providing getter methods.
- **AdminUser Class Encapsulation:**
  - Protect sensitive admin user information (e.g., admin level) by using private attributes and providing getter methods.

Polymorphism:

- **Polymorphic Methods:**
  - Implement polymorphic methods like `make_post(post_content)` in both the `User` and `AdminUser` classes. This method can be used to add a new post to the respective user's list of posts.

Use Cases:

1. **Make Posts:**
   - Create instances of the `User` and `AdminUser` classes to make posts.
2. **Remove Posts (Admin User):**
   - Use the `remove_post(post_content)` method to remove posts made by any user.
3. **Display Posts:**
   - Use the `display_posts()` method to display all posts made by a user.

Problem 4: Online Banking System

You are tasked with designing a comprehensive online banking system that includes various types of accounts, each with its unique features and functionalities. Implement the system using object-oriented programming (OOP) principles, ensuring proper inheritance, encapsulation, polymorphism, and adherence to the given specifications.

Classes:

1. **Account:**
   - Attributes:
     - `account_number`: Integer representing the unique identifier for the account.
     - `balance`: Float representing the current balance of the account.
   - Methods:
     - `deposit(amount)`: Add money to the account.
     - `withdraw(amount)`: Withdraw money from the account.
2. **SavingsAccount (Subclass of Account):**

- Attributes:
  - `interest_rate`: Float representing the interest rate for the savings account.
- Methods:
  - `calculate_interest()`: Calculate and add interest to the account balance.

3. **FixedAccount (Subclass of Account):**
   - Attributes:
     - `limit`: Float representing the withdrawal limit for the fixed account.
   - Methods:
     - `withdraw(amount)`: Override the `withdraw` method to enforce the withdrawal limit.
4. **CheckingAccount (Subclass of Account):**
   - Attributes:
     - `overdraft_limit`: Float representing the overdraft limit for the checking account.
   - Methods:
     - `withdraw(amount)`: Override the `withdraw` method to allow for limited overdraft.
5. **BusinessAccount (Subclass of Account):**
   - Attributes:
     - `transaction_fee`: Float representing the fee for each transaction in the business account.
   - Methods:
     - `withdraw(amount)`: Override the `withdraw` method to include transaction fees.

Inheritance:

- **Account Inheritance:**
  - Create subclasses for different types of accounts (e.g., `SavingsAccount`, `CheckingAccount`, `BusinessAccount`). Each subclass should inherit from the

`Account` class and may have additional attributes specific to the account type.

Encapsulation:

- **Account Class Encapsulation:**
  - Protect sensitive account information (e.g., balance) by using private attributes and providing getter and setter methods.
- **FixedAccount Class Encapsulation:**
  - Protect sensitive information (e.g., withdrawal limit) by using private attributes and providing getter and setter methods.

Polymorphism:

- **Polymorphic Methods:**
  - Implement polymorphic methods like `withdraw(amount)` in the `Account`, `FixedAccount`, `CheckingAccount`, and `BusinessAccount` classes. Each class should override the `withdraw` method to provide specific functionality.

Use Cases:

1. **Deposit and Withdraw:**
   - Create instances of the `Account`, `SavingsAccount`, `FixedAccount`, `CheckingAccount`, and `BusinessAccount` classes to deposit and withdraw money.
2. **Calculate Interest (Savings Account):**
   - Use the `calculate_interest()` method to add interest to the savings account balance.
3. **Enforce Withdrawal Limit (Fixed Account):**
   - Use the overridden `Withdraw` method to enforce the withdrawal limit for the fixed account.
4. **Overdraft Limit (Checking Account):**
   - Use the overridden `Withdraw` method to allow limited overdraft for the checking account.
5. **Transaction Fee (Business Account):**

- Use the overridden `withdraw` method to include transaction fees for the business account.

Problem 5: Shape Hierarchy

Design a system to represent different geometric shapes with the following classes:

Classes:

1. **Shape (Abstract Class):**
   - Abstract Method:
     - `calculate_area()`: An abstract method to calculate the area of the shape.
2. **Circle (Subclass of Shape):**
   - Attributes:
     - `radius`: Float representing the radius of the circle.
   - Methods:
     - `calculate_area()`: Override the abstract method to calculate the area of the circle.
3. **Rectangle (Subclass of Shape):**
   - Attributes:
     - `length`: Float representing the length of the rectangle.
     - `width`: Float representing the width of the rectangle.
   - Methods:
     - `calculate_area()`: Override the abstract method to calculate the area of the rectangle.

Abstract Method:

- **Shape Class Abstract Method:**
  - The `calculate_area()` method is declared as an abstract method in the `Shape` class. Each subclass must provide its own implementation for calculating the area.

Method Overriding:

- **Circle and Rectangle:**
  - Both the `Circle` and `Rectangle` classes override the `calculate_area()` method inherited from the abstract `Shape` class. Each subclass provides its specific implementation to calculate the area based on its attributes.

Problem 6: Complex Number Operations

Design a system to perform operations on complex numbers with the following classes:

Classes:

1. **ComplexNumber:**
   - Attributes:
     - `real_part`: Float representing the real part of the complex number.
     - `imaginary_part`: Float representing the imaginary part of the complex number.
   - Methods:
     - `__add__(other)`: Override the addition operator to add two complex numbers.
     - `__sub__(other)`: Override the subtraction operator to subtract one complex number from another.
     - `__mul__(other)`: Override the multiplication operator to multiply two complex numbers.
     - `__str__()`: Override the string representation for better display.

Operator Overriding:

- **ComplexNumber Class Operator Overriding:**
  - The `__add__`, `__sub__`, and `__mul__` methods are overridden to perform addition, subtraction, and multiplication of complex numbers, respectively.

Problem 7: Employee Management System

Design a simple employee management system with the following classes:

Classes:

1. **Employee:**
   - Attributes:
     - `employee_id`: Integer representing the unique identifier for the employee.
     - `name`: String representing the name of the employee.
     - `salary`: Float representing the salary of the employee.
   - Methods:
     - `__str__()`: Override the string representation for better display.
2. **Manager (Subclass of Employee):**
   - Attributes:
     - `bonus`: Float representing the bonus amount for the manager.
   - Methods:
     - `__str__()`: Override the string representation to include bonus information.

Encapsulation:

- **Employee Class Encapsulation:**
  - The attributes `employee_id`, `name`, and `salary` are encapsulated using private attributes, and getter methods (`get_employee_id()`, `get_name()`, `get_salary()`) are provided to access them.
- **Manager Class Encapsulation:**
  - The attribute `bonus` is encapsulated using a private attribute, and a getter method (`get_bonus()`) is provided to access it.
  -

Problem 8: Animal Kingdom

Design a class hierarchy to represent different animals with the following classes:

Classes:

1. **Animal (Abstract Class):**
   - Abstract Method:
     - `make_sound()`: An abstract method to make a sound.
2. **Mammal (Subclass of Animal):**
   - Methods:
     - `give_birth()`: Method to represent the mammal giving birth.
3. **Bird (Subclass of Animal):**
   - Methods:
     - `lay_eggs()`: Method to represent the bird laying eggs.

Abstract Method:

- **Animal Class Abstract Method:**
  - The `make_sound()` method is declared as an abstract method in the `Animal` class. Each subclass must provide its own implementation for making a sound.

Polymorphism:

- **Polymorphic Methods:**
  - Implement polymorphic methods like `make_sound()` in both the `Mammal` and `Bird` classes. Each subclass provides its specific implementation for making a sound.

Complex Problem 1: Company Hierarchy

Design a complex system to represent the hierarchy of employees in a large company with multiple departments.

Classes:

1. **Employee:**
   - Attributes:

    - `employee_id`: Integer representing the unique identifier for the employee.
    - `name`: String representing the name of the employee.
    - `position`: String representing the job position of the employee.
  - Methods:
    - `get_details()`: Display detailed information about the employee.

2. **Manager (Subclass of Employee):**
  - Attributes:
    - `team_members`: List of instances of the `Employee` class representing the employees managed by the manager.
    - `department`: String representing the department managed by the manager.
  - Methods:
    - `add_team_member(employee)`: Add a new team member to the manager's team.
    - `remove_team_member(employee)`: Remove a team member from the manager's team.
3. **CEO (Subclass of Manager):**
  - Attributes:
    - `direct_reports`: List of instances of the `Manager` class representing the managers directly reporting to the CEO.
  - Methods:
    - `add_direct_report(manager)`: Add a new manager to directly report to the CEO.
    - `remove_direct_report(manager)`: Remove a manager from the direct reports.

Inheritance:

- **Employee Inheritance:**
  - The `Manager` and `CEO` classes inherit from the `Employee` class.
- **Manager Inheritance:**

- The `CEO` class inherits from the `Manager` class.

Complex Hierarchy and Operations:

- **Multiple Levels of Hierarchy:**
  - Implement a hierarchy with employees, managers, and the CEO, representing a realistic organizational structure.
- **Hierarchical Operations:**
  - Implement operations for adding and removing team members for managers and adding and removing direct reports for the CEO.
- **Display Hierarchy:**
  - Implement a method to display the entire organizational hierarchy, showing the relationships between employees, managers, and the CEO.

Complex Problem 2: University Course Management

Design a system to represent the complex structure of university courses, including departments, professors, and students.

Classes:

1. **Person:**
   - Attributes:
     - `person_id`: Integer representing the unique identifier for the person.
     - `name`: String representing the name of the person.
   - Methods:
     - `get_details()`: Display detailed information about the person.
2. **Professor (Subclass of Person):**
   - Attributes:
     - `professor_id`: Integer representing the unique identifier for the professor.
     - `department`: String representing the department to which the professor belongs.

- `courses_taught`: List of instances of the `Course` class representing the courses taught by the professor.

- Methods:
  - `add_course(course)`: Add a course to the professor's list of courses.
  - `remove_course(course)`: Remove a course from the professor's list.

3. **Student (Subclass of Person):**
   - Attributes:
     - `student_id`: Integer representing the unique identifier for the student.
     - `courses_enrolled`: List of instances of the `Course` class representing the courses enrolled by the student.
   - Methods:
     - `enroll_course(course)`: Enroll in a new course.
     - `withdraw_course(course)`: Withdraw from a course.
4. **Department:**
   - Attributes:
     - `department_name`: String representing the name of the department.
     - `professors`: List of instances of the `Professor` class representing the professors in the department.
     - `courses_offered`: List of instances of the `Course` class representing the courses offered by the department.
5. **Course:**
   - Attributes:
     - `course_code`: String representing the unique code for the course.
     - `course_name`: String representing the name of the course.
     - `professor`: Instance of the `Professor` class representing the professor teaching the course.
     - `students_enrolled`: List of instances of the `Student` class representing the students enrolled in the course.

Inheritance and Complex Relationships:

- **Inheritance:**
  - The `Professor` and `Student` classes inherit from the `Person` class.
- **Complex Relationships:**
  - Establish complex relationships between departments, professors, students, and courses, representing the intricate structure of a university.

Operations and Interactions:

- **Enroll and Withdraw Operations:**
  - Implement methods for students to enroll in and withdraw from courses.
- **Add and Remove Course Operations:**
  - Implement methods for professors to add and remove courses they teach.
- **Display University Hierarchy:**
  - Implement a method to display the entire university hierarchy, showing the relationships between departments, professors, students, and courses.

Complex Problem 3: Geometric Shape Operations

Design a system to perform operations on geometric shapes with a focus on abstract methods, method overriding, and operator overloading.

Classes:

1. **Shape (Abstract Class):**
   - Abstract Method:
     - `calculate_area()`: An abstract method to calculate the area of the shape.
2. **Rectangle (Subclass of Shape):**
   - Attributes:
     - `length`: Float representing the length of the rectangle.
     - `width`: Float representing the width of the rectangle.

- Methods:
  - `calculate_area()`: Override the abstract method to calculate the area of the rectangle.

3. **Circle (Subclass of Shape):**
   - Attributes:
     - `radius`: Float representing the radius of the circle.
   - Methods:
     - `calculate_area()`: Override the abstract method to calculate the area of the circle.

Operator Overloading:

- **Rectangle and Circle Classes:**
  - Overload the `+` operator to create a new shape by combining two existing shapes (e.g., adding the areas of two shapes).

Abstract Method, Method Overriding, and Operator Overloading:

- **Abstract Method:**
  - The `calculate_area()` method is declared as an abstract method in the `Shape` class. Each subclass must provide its own implementation for calculating the area.
- **Method Overriding:**
  - Both the `Rectangle` and `Circle` classes override the `calculate_area()` method inherited from the abstract `Shape` class. Each subclass provides its specific implementation to calculate the area based on its attributes.
- **Operator Overloading:**
  - The `Rectangle` and `Circle` classes overload the `+` operator to allow the combination of two shapes, resulting in a new shape with a combined area.

Complex Problem 4: Matrix Operations

Design a system to perform operations on matrices with a focus on abstract methods, method overriding, and operator overloading.

Classes:

1. **Matrix (Abstract Class):**
   - Attributes:
     - `rows`: Integer representing the number of rows in the matrix.
     - `columns`: Integer representing the number of columns in the matrix.
     - `data`: 2D list representing the elements of the matrix.
   - Abstract Method:
     - `multiply_scalar(scalar)`: An abstract method to multiply the matrix by a scalar.
2. **SquareMatrix (Subclass of Matrix):**
   - Methods:
     - `calculate_determinant()`: Method to calculate the determinant of the square matrix.

Operator Overloading:

- **Matrix and SquareMatrix Classes:**
  - Overload the `@` operator to perform matrix multiplication between two matrices.
  - Overload the `*` operator to perform exponentiation of a square matrix by a scalar.

Abstract Method, Method Overriding, and Operator Overloading:

- **Abstract Method:**
  - The `multiply_scalar()` method is declared as an abstract method in the `Matrix` class. Each subclass must provide its own implementation for multiplying the matrix by a scalar.
- **Method Overriding:**
  - The `SquareMatrix` class overrides the `multiply_scalar()` method inherited from the abstract `Matrix` class. It provides a specific implementation for multiplying a square matrix by a scalar.

- **Operator Overloading:**
  - The `Matrix` and `SquareMatrix` classes overload the `*` operator to perform matrix multiplication between two matrices. The `SquareMatrix` class also overloads the `**` operator for exponentiation.

Comprehensive Problem: Enterprise Resource Management System

Design a sophisticated Enterprise Resource Management (ERM) system that encompasses various aspects of a large organization, including employees, projects, financials, and interactions.

Classes:

1. **Person:**
   - Attributes:
     - `person_id`: Integer representing the unique identifier for the person.
     - `name`: String representing the name of the person.
     - `contact_info`: String representing the contact information of the person.
   - Methods:
     - `get_details()`: Display detailed information about the person.
2. **Employee (Subclass of Person):**
   - Attributes:
     - `employee_id`: Integer representing the unique identifier for the employee.
     - `position`: String representing the job position of the employee.
     - `projects_assigned`: List of instances of the `Project` class representing the projects assigned to the employee.
   - Methods:
     - `assign_project(project)`: Assign a new project to the employee.
     - `complete_project(project)`: Mark a project as completed.

3. **Manager (Subclass of Employee):**
    - Attributes:
        - `team_members`: List of instances of the `Employee` class representing the employees managed by the manager.
        - `department`: String representing the department managed by the manager.
        - `budget_allocated`: Float representing the budget allocated to the manager's department.
    - Methods:
        - `add_team_member(employee)`: Add a new team member to the manager's team.
        - `remove_team_member(employee)`: Remove a team member from the manager's team.
        - `allocate_budget(amount)`: Allocate a budget to the manager's department.
4. **CEO (Subclass of Manager):**
    - Attributes:
        - `direct_reports`: List of instances of the `Manager` class representing the managers directly reporting to the CEO.
    - Methods:
        - `add_direct_report(manager)`: Add a new manager to directly report to the CEO.
        - `remove_direct_report(manager)`: Remove a manager from the direct reports.
5. **Project:**
    - Attributes:
        - `project_code`: String representing the unique code for the project.
        - `project_name`: String representing the name of the project.
        - `project_manager`: Instance of the `Manager` class representing the manager overseeing the project.

- `start_date`: Date representing the start date of the project.
- `end_date`: Date representing the end date of the project (if applicable).

6. **FinancialSystem:**
   - Attributes:
     - `total_budget`: Float representing the total budget for the organization.
     - `expenses`: Float representing the total expenses incurred.
     - `profits`: Float representing the total profits generated.
   - Methods:
     - `calculate_profits()`: Calculate and update the total profits based on project completions.

Inheritance, Polymorphism, Encapsulation, and Complex Relationships:

- **Inheritance:**
  - The `Employee`, `Manager`, and `CEO` classes inherit from the `Person` class.
- **Polymorphism:**
  - Implement polymorphic methods like `get_details()`, `assign_project()`, and `allocate_budget()` to handle interactions with employees, managers, and the CEO.
- **Encapsulation:**
  - Protect sensitive information (e.g., employee IDs, project codes) by using private attributes and providing getter methods.
- **Complex Relationships:**
  - Establish complex relationships between CEOs, managers, employees, and projects, representing the intricate structure of a large organization.

Operations and Interactions:

- **Assign and Complete Project Operations:**
  - Implement methods for employees to be assigned projects and mark them as completed.

- **Team Management Operations:**
  - Implement methods for managers to add and remove team members.
- **Financial Operations:**
  - Implement methods to calculate and update profits based on project completions.
- **Display Organization Hierarchy:**
  - Implement a method to display the entire organizational hierarchy, showing the relationships between CEOs, managers, employees, projects, and financials.

Command-Line Virtual Pet Simulation Game

You are tasked with developing a command-line-based virtual pet simulation game. The game will allow users to adopt and interact with virtual pets, each with its unique characteristics. The game should be implemented using object-oriented programming (OOP) concepts, including inheritance, polymorphism, encapsulation, abstract methods, method overriding, and operator overloading.

Classes:

1. **Pet:**
   - Abstract class representing a generic pet.
   - Attributes:
     - `pet_id`: Integer representing the unique identifier for the pet.
     - `name`: String representing the name of the pet.
     - `happiness`: Float representing the happiness level of the pet.
     - `hunger`: Float representing the hunger level of the pet.
   - Methods:
     - `play()`: Abstract method to simulate playing with the pet. Implemented in subclasses.
     - `feed()`: Abstract method to simulate feeding the pet. Implemented in subclasses.
     - `get_details()`: Display detailed information about the pet.

2. **Dog (Subclass of Pet):**
    - Represents a virtual dog with specific play and feed behaviors.
    - Methods:
        - `play()`: Override the `play()` method to simulate playing fetch with the dog.
        - `feed()`: Override the `feed()` method to simulate feeding the dog with dog food.
3. **Cat (Subclass of Pet):**
    - Represents a virtual cat with specific play and feed behaviors.
    - Methods:
        - `play()`: Override the `play()` method to simulate playing with a laser pointer.
        - `feed()`: Override the `feed()` method to simulate feeding the cat with cat food.
4. **Owner:**
    - Represents the virtual pet owner.
    - Attributes:
        - `owner_id`: Integer representing the unique identifier for the owner.
        - `pets`: List of instances of the `Pet` class representing the owner's pets.
    - Methods:
        - `adopt_pet(pet)`: Adopt a new pet and add it to the owner's list of pets.
        - `play_with_pet(pet)`: Invoke the `play()` method of a specific pet.
        - `feed_pet(pet)`: Invoke the `feed()` method of a specific pet.

Inheritance, Polymorphism, and Encapsulation:

- **Inheritance:**
    - The `Dog` and `Cat` classes inherit from the `Pet` class.
- **Polymorphism:**

- Implement polymorphic methods like `play_with_pet()` and `feed_pet()` in the `Owner` class to interact with different types of pets.

- **Encapsulation:**
  - Protect sensitive information (e.g., pet IDs) by using private attributes and providing getter methods.

Game Interactions and Complexity:

- **Adopt and Care for Pets:**
  - Simulate the adoption of pets by an owner and the subsequent care activities.
- **Simulate Pet Activities:**
  - Simulate playing and feeding activities for both dogs and cats, each with their unique characteristics.
- **Owner-Pet Relationships:**
  - Represent complex relationships between owners and their pets in the game.

Bonus: Operator Overloading and Game Dynamics

- **Operator Overloading:**
  - Overload the `+` operator to simulate breeding, producing a new pet with a combination of characteristics from two existing pets.
- **Game Dynamics:**
  - Implement game dynamics that affect the pet's happiness and hunger levels over time. The owner's actions impact the pet's well-being, and the game should have a scoring mechanism based on how well the pets are cared for.

Instructions:

1. Implement the classes and methods outlined above using Python.
2. Create a command-line interface that allows users to interact with the virtual pets.
3. Provide clear instructions and options for the user to play, feed, view pet details, and exit the game.

4. Ensure that the game dynamics are engaging, and the interactions reflect the care and well-being of the virtual pets.

"May your code be bug-free, your logic flawless, and your journey in the realms of programming be a constant source of inspiration. Happy coding!"