ChatGPT

Object-Oriented Design: Mapping User Actions to Classes (Lesson Notes)

Object-oriented design begins by identifying the **nouns** and **verbs** in a software project's requirements. Nouns often correspond to **entities or roles** (which become classes or objects with properties), and verbs correspond to **actions** (which become methods/functions) [1] [2]. In other words, *nouns are properties and verbs are methods* [1]. By listening to the problem description or use cases, we determine what real-world things (users, items, etc.) need to be represented as classes and what actions they perform (or are performed on them) as methods [3] [4]. Data attributes (often described by adjectives or details) become the **properties** of those classes [2].

Below, we will walk through five example projects and demonstrate how to identify user personas (roles) and system entities, map their actions to class methods (verbs to functions), and their data to class properties (nouns to variables). For each project, we outline the key roles and actions, then present example Java classes with properties and methods (with minimal or no implementation, focusing on design). We also include UML-style representations (diagrams or tables) to illustrate the relationships.

1. Bank Management System

Identifying Roles and Actions: In a bank management system, common actors include bank customers and bank staff. Key actions in this domain are things like *opening accounts, depositing money, withdrawing money,* and *checking balances*. We identify the primary entities (nouns) as **Bank**, **Customer**, **Account**, and possibly **Transaction** [5]. For example, a typical scenario might be: *A Customer goes to the Bank and withdraws Money* – from this we deduce that **Customer** and **Account** are classes, `withdraw` is a method, and **Money/Balance** is a property [6]. The Bank itself can be a class that oversees accounts.

Mapping Actions to Classes: Consider the actions and how they map to class methods and properties:

Actor/Role	Action (Verb)	Class & Method	Associated Data (Property)
Customer (Account Holder)	Deposit money	`BankAccount.deposit(amount)`	`balance` (increases)
Customer (Account Holder)	Withdraw money	`BankAccount.withdraw(amount)`	`balance` (decreases)
Bank Staff/ System	Open new account	`Bank.openAccount(Customer)`	`accounts` list (stored in Bank)
Customer/Bank System	Check account balance	`BankAccount.getBalance()`	`balance` (returned/read)

In this mapping, **BankAccount** (or simply **Account**) is a class with a `balance` property and methods like `deposit()` and `withdraw()`. A **Bank** class might hold a collection of accounts and have methods to create accounts or compute totals. A **Customer** class could represent a person with one or more accounts.

UML class diagram for a banking system. Classes like Bank, Account, Customer, and Transaction are common. Bank aggregates multiple Account objects, each Account is associated with a Customer and possibly multiple Transaction records [5] .

Example Classes (Java): Below are simplified Java classes for a banking system. We define a `Bank` class to manage accounts, a `Customer` class for bank customers, and a `BankAccount` class for accounts. Notice how methods correspond to actions (verbs) and attributes correspond to data (nouns). We also demonstrate one class calling another's methods (e.g., `Bank.getTotalAssets()` calling each account's `getBalance()`):

```
// BankAccount.java
class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double initialDeposit) {
        this.accountNumber = accountNumber;
        this.balance = initialDeposit;
    }

    public void deposit(double amount) {
        // Increase balance by amount
        balance += amount;
        System.out.println("Deposited $" + amount + " into Account " +
accountNumber);
    }

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrew $" + amount + " from Account " +
accountNumber);
        } else {
            System.out.println("Withdrawal of $" + amount + " failed:
Insufficient funds.");
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

```java
// Customer.java
class Customer {
    private String name;
    // A customer could have multiple accounts; for simplicity, one primary
account:
    private BankAccount account;

    public Customer(String name) {
        this.name = name;
    }

    public void setAccount(BankAccount account) {
        this.account = account;
    }

    public BankAccount getAccount() {
        return account;
    }

    public String getName() {
        return name;
    }
}
```

```java
// Bank.java
import java.util.ArrayList;
import java.util.List;

class Bank {
    private String bankName;
    private List<BankAccount> accounts = new ArrayList<>();

    public Bank(String bankName) {
        this.bankName = bankName;
    }

    public BankAccount openAccount(Customer customer, String accountNumber,
double initialDeposit) {
        // Create a new account and associate it with the customer
        BankAccount newAccount = new BankAccount(accountNumber,
initialDeposit);
        accounts.add(newAccount);
        customer.setAccount(newAccount);
        System.out.println("Opened new account " + accountNumber + " for " +
customer.getName());
        return newAccount;
    }

    public double getTotalAssets() {
```

```
        // Calculate total balance across all accounts
        double total = 0;
        for (BankAccount acct : accounts) {
            total += acct.getBalance();  // calling another object's method
        }
        return total;
    }
}
```

In this design, **Customer** has a reference to a **BankAccount**. The **Bank** manages a list of accounts. Methods `deposit()` and `withdraw()` on `BankAccount` represent actions that change the account's state (`balance`). The `Bank.getTotalAssets()` method iterates over accounts and calls each account's `getBalance()` – this demonstrates **interaction between objects** (Bank using Account's methods). For example, if a customer Alice opens an account and deposits money:

```
Bank bank = new Bank("MyBank");
Customer alice = new Customer("Alice");
BankAccount aliceAcct = bank.openAccount(alice, "A1001", 500.0);
aliceAcct.deposit(200.0);
aliceAcct.withdraw(100.0);
System.out.println("Bank total assets: $" + bank.getTotalAssets());
```

This sequence would create an account for Alice with $500, deposit $200 (balance becomes $700), withdraw $100 (balance $600), and then the bank's total assets would reflect Alice's account balance of $600.

2. Online Retail Store (E-Commerce)

Identifying Roles and Actions: An online retail system typically involves **Customers** (buyers) and possibly **Admins/Sellers** who manage the product catalog [7] [8]. Key actions include *browsing products, adding items to a shopping cart, removing items, and checking out (placing orders)*. The main entities (nouns) we identify are **Product**, **ShoppingCart** (or Cart), **Order**, and **Customer** [9]. For instance, a Customer will *view Products*, *add Product to Cart*, and *place an Order*. An Admin (seller) might *add or remove products* from the catalog [8], though we will focus on the customer actions for simplicity.

Mapping Actions to Classes: From the use cases, we map verbs to methods in our classes:

- **Customer** – actions: *view products, add to cart, remove from cart, checkout.* These can be methods like `addToCart(product)`, `removeFromCart(product)`, or possibly interactions through a `Cart` object's methods.
- **ShoppingCart** – actions: *calculate total, list items, clear cart on checkout*. Methods could include `addProduct(product, qty)`, `removeProduct(product)`, `calculateTotal()` etc.
- **Admin** – actions: *add new product, update or delete product.* This suggests methods like `addProduct(Product)` in a Catalog or Shop class (or Admin class).
- **Order** – represents the result of checkout; an `Order` class might have properties for ordered items and total amount, and perhaps a method `confirm()`.

Based on a typical e-commerce class model: *"Customer, Order, Product, and Payment"* are common classes; a **Customer** may have an association with **Order** (customers place orders), an **Order** contains multiple **Product** items, and there may be a **Payment** associated with an Order [9]. In our example, we will implement Customer, Product, Cart, and Order (covering payment conceptually via an Order's total).

Example Classes (Java): Below, we design classes for a simple online store: `Product`, `ShoppingCart` (with methods to add/remove products), `Customer` (with a shopping cart), and `Order` (to record a completed purchase). We demonstrate interactions such as a customer using the cart's methods and the cart calculating the total by accessing each product's price.

```
// Product.java
class Product {
    private String id;
    private String name;
    private double price;

    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
    public String getId() { return id; }
    public String getName() { return name; }
    public double getPrice() { return price; }
}
```

```
// ShoppingCart.java
import java.util.ArrayList;
import java.util.List;

class ShoppingCart {
    private List<Product> items = new ArrayList<>();

    public void addProduct(Product product, int quantity) {
        // Add the product to the cart 'quantity' times
        for (int i = 0; i < quantity; i++) {
            items.add(product);
        }
        System.out.println("Added " + quantity + " of " + product.getName()
+ " to cart.");
    }

    public void removeProduct(String productId, int quantity) {
        // Remove up to 'quantity' instances of the product from the cart
        int removedCount = 0;
        for (int i = 0; i < items.size() && removedCount < quantity; ) {
            if (items.get(i).getId().equals(productId)) {
                items.remove(i);
                removedCount++;
```

```
            } else {
                i++;
            }
        }
        System.out.println("Removed " + removedCount + " of product " +
productId + " from cart.");
    }

    public double calculateTotal() {
        double total = 0;
        for (Product p : items) {
            total += p.getPrice();
        }
        return total;
    }

    public void viewCart() {
        System.out.println("Current items in cart:");
        if (items.isEmpty()) {
            System.out.println("  (cart is empty)");
            return;
        }
        // Tally quantities by product for display:
        java.util.Map<String, Integer> countMap = new java.util.HashMap<>();
        for (Product p : items) {
            countMap.put(p.getName(), countMap.getOrDefault(p.getName(), 0)
+ 1);
        }
        for (String productName : countMap.keySet()) {
            System.out.println("  " + productName + " x " +
countMap.get(productName));
        }
    }

    public void clearCart() {
        items.clear();
    }
}
```

```
// Customer.java
class Customer {
    private String name;
    private ShoppingCart cart = new ShoppingCart();  // each customer has a
cart

    public Customer(String name) {
        this.name = name;
    }
    public ShoppingCart getCart() {
        return cart;
```

```
    }
    public String getName() {
        return name;
    }

    // The Customer can perform higher-level actions using the cart
    public Order checkout() {
        double totalAmount = cart.calculateTotal();
        Order order = new Order(this, cart, totalAmount);
        cart.clearCart();
        System.out.println(name + " checked out. Order total: $" +
totalAmount);
        return order;
    }
}
```

```
// Order.java
import java.util.List;

class Order {
    private static int nextOrderId = 1;  // simple auto-increment for IDs
    private int orderId;
    private Customer customer;
    private List<Product> items;
    private double totalAmount;

    public Order(Customer customer, ShoppingCart cart, double totalAmount) {
        this.orderId = nextOrderId++;
        this.customer = customer;
        // Copy items from cart to order (shallow copy for simplicity)
        this.items = new ArrayList<>(/* package-private */ cart.items);
        this.totalAmount = totalAmount;
    }

    public int getOrderId() { return orderId; }
    public Customer getCustomer() { return customer; }
    public double getTotalAmount() { return totalAmount; }
    // (Other methods like getItems() could be here, omitted for brevity)
}
```

In this design: - **Product** holds data (id, name, price). - **ShoppingCart** holds a collection of Products and provides methods `addProduct`, `removeProduct`, `viewCart`, `calculateTotal`. These methods are the behaviors (verbs) corresponding to cart management actions. - **Customer** uses a ShoppingCart; the `checkout()` method in `Customer` calls `cart.calculateTotal()` and then creates an **Order** object. This demonstrates a method in one class (`Customer.checkout`) calling methods of another (`ShoppingCart.calculateTotal`) and interacting with another object (creating an `Order`). The `Order` class simply captures the outcome of a checkout (which customer, what items, total cost).

For example, a customer using these classes:

```
Customer bob = new Customer("Bob");
Product laptop = new Product("P100", "Laptop", 800.00);
Product phone = new Product("P200", "Smartphone", 500.00);

bob.getCart().addProduct(laptop, 1);
bob.getCart().addProduct(phone, 2);
bob.getCart().viewCart();           // shows Laptop x1, Smartphone x2
Order order = bob.checkout();       // calculates total and clears cart
System.out.println("Order ID " + order.getOrderId() + " for " +
order.getCustomer().getName() +
                   " has total $" + order.getTotalAmount());
```

This might output a summary of Bob's cart contents, followed by an order confirmation indicating Bob's order total (in this case $800 + 2*$500 = $1800).

3. Hospital Management System

Identifying Roles and Actions: In a hospital management context, the key user personas and entities include **Patients**, **Doctors**, and possibly **Administrative Staff** (like a receptionist or hospital admin). Important actions include *registering patients, scheduling appointments, conducting consultations,* etc. The primary classes (nouns) we can extract are **Hospital**, **Patient**, **Doctor**, and **Appointment** [10]. Often a generic **Person** class is used as a base for Patient and Doctor (since both share common properties like name, age) [11]. For example, a *Receptionist schedules an Appointment between a Patient and a Doctor*. Here **Appointment** is a class linking a patient and doctor (with a date/time), and scheduling is a method that likely lives in a Hospital or Scheduling class.

In one class diagram example, the classes included *Hospital, Reception, Patient, Doctor,* and *Report*, where **Hospital** manages doctors and patients, **Reception** schedules appointments with doctors, and **Patient** and **Doctor** have relationships through appointments. We will simplify to focus on Patient–Doctor appointment scheduling.

Mapping Actions to Classes:

- **Patient** – actions: *book appointment, receive treatment*. (Booking could be a method like `requestAppointment` or done via a Hospital system.)
- **Doctor** – actions: *check availability, treat patient, write report*. Methods could include `isAvailable(date)` or `addAppointment(Appointment)`.
- **Receptionist/System** – actions: *schedule appointment*. We might implement this as a method in a **Hospital** class like `scheduleAppointment(patient, doctor, date)` which creates an Appointment.
- **Appointment** – an entity representing the meeting; properties: date/time, and methods like `getDetails()`.

Example Classes (Java): We demonstrate a base class `Person` (with common attributes) and two subclasses `Patient` and `Doctor`. We also have an `Appointment` class and a `Hospital` class that schedules appointments. This design uses **inheritance** (Patient and Doctor inherit from Person) to avoid duplication of common properties [11]. The `Hospital.scheduleAppointment` method illustrates a system action that involves multiple classes (it checks a Doctor's availability and then creates an Appointment linking the Doctor and Patient):

```java
// Person.java - a base class for common person attributes
class Person {
    protected String name;
    protected int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```java
// Patient.java - inherits from Person
class Patient extends Person {
    private String patientId;
    public Patient(String name, int age, String patientId) {
        super(name, age);
        this.patientId = patientId;
    }
    public String getPatientId() { return patientId; }
}
```

```java
// Doctor.java - inherits from Person
class Doctor extends Person {
    private String specialization;
    // For simplicity, track availability in a basic way (e.g., always
available or fixed hours)
    public Doctor(String name, int age, String specialization) {
        super(name, age);
        this.specialization = specialization;
    }
    public String getSpecialization() { return specialization; }

    public boolean isAvailable(String date) {
        // In a real system, check doctor's schedule. Here, always return
true for demo.
        System.out.println(name + " is available on " + date);
        return true;
    }

    public void seePatient(Patient patient) {
        // Simulate a doctor treating a patient
        System.out.println("Dr. " + name + " is seeing patient " +
patient.getName());
    }
}
```

```
// Appointment.java
class Appointment {
    private Patient patient;
    private Doctor doctor;
    private String date;  // date/time as string for simplicity

    public Appointment(Patient patient, Doctor doctor, String date) {
        this.patient = patient;
        this.doctor = doctor;
        this.date = date;
    }
    public String getDetails() {
        return "Appointment on " + date + " with Dr. " + doctor.getName() +
" for patient " + patient.getName();
    }
}
```

```
// Hospital.java
class Hospital {
    private String name;
    public Hospital(String name) {
        this.name = name;
    }

    public Appointment scheduleAppointment(Patient patient, Doctor doctor,
String date) {
        // Check if doctor is available, then create appointment
        if (doctor.isAvailable(date)) {  // calls Doctor's method
            Appointment appt = new Appointment(patient, doctor, date);
            System.out.println("Scheduled: " + appt.getDetails());
            return appt;
        } else {
            System.out.println("Could not schedule appointment: Doctor not
available.");
            return null;
        }
    }
}
```

In this design: - **Person** is a general class for any person in the system, with properties like name and age. - **Patient** and **Doctor** extend Person, adding specific fields (`patientId`, `specialization`) and behaviors. This is an example of using inheritance to model specialized roles in the system. - **Doctor** has a method `isAvailable(date)` representing an action to check schedule (here simplified to always true) and a method `seePatient()` to represent treating a patient. - **Appointment** is a simple class linking a Patient and Doctor on a given date. - **Hospital** provides a high-level action `scheduleAppointment` which internally calls `doctor.isAvailable()` (method call across classes) and if true, creates an `Appointment`. The appointment creation bundles the data (patient, doctor, date) into one object.

For example, using these classes:

```
Hospital hospital = new Hospital("City Hospital");
Patient john = new Patient("John Doe", 30, "P1001");
Doctor drSmith = new Doctor("Smith", 45, "Cardiology");

hospital.scheduleAppointment(john, drSmith, "2025-09-01 10:00 AM");
// Output might include: "Dr. Smith is available on 2025-09-01 10:00 AM"
// and then "Scheduled: Appointment on 2025-09-01 10:00 AM with Dr. Smith for
patient John Doe".
drSmith.seePatient(john);
// Output: "Dr. Smith is seeing patient John Doe"
```

This sequence shows the Hospital scheduling an appointment (using the Doctor and Patient classes) and the Doctor subsequently seeing the patient. The **associations** here are clear: Hospital coordinates between Patient and Doctor, and an Appointment associates a Patient with a Doctor on a date [10].

UML class diagram for a hospital management system. Common classes include Hospital, Patient, Doctor, and Appointment, with Hospital aggregating multiple Patient and Doctor instances. Patient and Doctor are often linked through Appointment entries [10]. (This diagram also shows inheritance of roles, e.g., Person as a base class for staff and patients.)

4. Grievance/Complaints Management System

Identifying Roles and Actions: In a complaints management system, the typical users are **Customers** (or citizens/users who file complaints) and **Staff/Admin** who handle those complaints [12]. The system itself may have components for tracking and notifying. Key actions include *submitting a complaint, assigning it to staff, updating status (resolving or closing the complaint),* and *viewing status updates*. Core entities (classes) to model are **Complaint**, **User** (with specialized roles like Customer and Staff/Admin), and possibly **Attachment** (if complaints can have files) [13].

According to a design outline, **User**, **Complaint**, and **Admin** are key classes, where **Customer** and **Staff** can be subclasses of User (to represent specialized roles) [14]. A **User** might have methods to submit a complaint, and an **Admin/Staff** user would have methods to resolve or update complaints. The **Complaint** class holds details like a description, status, and possibly methods like `resolve()` to mark it resolved. For example, a *Customer submits a Complaint* (creating a Complaint record) and an *Admin resolves the Complaint* (changing its status) [13].

Mapping Actions to Classes:

- **Customer (a type of User)** – action: *submit complaint*. Method: e.g., `submitComplaint(description)` which creates a Complaint instance.
- **Staff/Admin (a type of User)** – action: *resolve complaint*. Method: e.g., `resolveComplaint(Complaint)` which updates the complaint's status.
- **System** – actions: *assign complaint, notify updates*. (These might be handled via an Admin user or a separate component; for our purposes, we focus on user actions.)
- **Complaint** – no independent actions as a user, but it has its own behavior: perhaps a method `markResolved()` to change its status internally.

Example Classes (Java): We model a base class `User` and two subclasses `Customer` and `Staff` (to represent an admin/staff user). The `Customer` class has a method to file a new complaint. The `Staff` class has a method to resolve a complaint. The `Complaint` class holds the complaint details and a status property. This setup uses inheritance for user roles (demonstrating how different user types share common attributes but have different actions) [13] :

```
// User.java – base class for any user in the system
class User {
    protected String name;
    protected String email;
    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
    public String getName() { return name; }
}
```

```
// Customer.java – a user who can submit complaints
class Customer extends User {
    public Customer(String name, String email) {
        super(name, email);
    }
    public Complaint submitComplaint(String description) {
        // Create a new complaint by this customer
        Complaint c = new Complaint(this, description);
        System.out.println(name + " submitted a complaint: " + description);
        return c;
    }
}
```

```
// Staff.java – a user who can resolve complaints (admin role)
class Staff extends User {
    public Staff(String name, String email) {
        super(name, email);
    }
    public void resolveComplaint(Complaint complaint) {
        // Mark the complaint as resolved
        complaint.markResolved();
        System.out.println("Staff " + name + " resolved complaint ID " +
complaint.getId());
    }
}
```

```
// Complaint.java
class Complaint {
    private static int nextId = 1;
    private int id;
```

```
    private User submitter;      // the user who submitted the complaint
    private String description;
    private String status;       // e.g., "Open", "Resolved"

    public Complaint(User submitter, String description) {
        this.id = nextId++;
        this.submitter = submitter;
        this.description = description;
        this.status = "Open";
    }
    public int getId() { return id; }
    public String getDescription() { return description; }
    public String getStatus() { return status; }

    public void markResolved() {
        status = "Resolved";
        // perhaps log resolution time, etc. (omitted for brevity)
    }
}
```

In this design: - **User** is a general class holding common user info (name, email). - **Customer** (extends User) has the ability to create a Complaint via `submitComplaint`. This method internally instantiates a new `Complaint` object, linking the complaint with the submitter (itself). - **Staff** (extends User) has the ability to resolve a complaint via `resolveComplaint`, which calls the complaint's own `markResolved()` method. This is an example of one object (`Staff`) invoking a method on another object (`Complaint`), i.e., the Staff uses the Complaint's functionality to change its state. - **Complaint** holds an ID, a reference to the submitting User, a description, and a status field. The method `markResolved()` changes the complaint's status to "Resolved".

For example usage:

```
Customer alice = new Customer("Alice", "alice@example.com");
Staff bob = new Staff("Bob", "bob@company.com");

Complaint comp = alice.submitComplaint("Internet not working in my area.");
System.out.println("Complaint Status: " + comp.getStatus());  // "Open"
bob.resolveComplaint(comp);
System.out.println("Complaint Status: " + comp.getStatus());  // "Resolved"
```

Here Alice submits a complaint (which is "Open" by default) and Bob, a staff member, resolves it. After Bob calls `resolveComplaint`, the complaint's status is "Resolved". This aligns with the real-world process: user files a complaint, admin/staff addresses it. The classes `Customer` and `Staff` are specialized from `User` (illustrating inheritance for roles) [15], and methods `submitComplaint()` and `resolveComplaint()` correspond to the actions of filing and resolving a complaint. The system could be extended with features like assigning complaints to specific staff, adding an **Attachment** class for documents (via aggregation with Complaint) [16], or notifications, but those are beyond this basic illustration.

5. Survey Application

Identifying Roles and Actions: In a survey application, there are typically two main user roles: **Survey Administrators** (or creators) who design and distribute surveys, and **Respondents/Users** who take the surveys. Sometimes there is also an **Analyst/Manager** role to analyze results [17]. Key actions include *creating a survey, adding questions to it, publishing or sharing the survey, filling out the survey (submitting responses),* and *analyzing the results*. The primary classes (nouns) likely include **Survey**, **Question**, **User** (with possibly subclasses like **Admin** and **Respondent**), and perhaps **Response** or **Result** to capture answers.

From the requirements perspective, we have: - **Administrator** – creates surveys and shares them [18]. - **User (Respondent)** – fills out surveys [19]. - **Manager/Analyst** – reviews or analyzes survey results [18]. - Each survey consists of multiple **Question** items.

A typical use-case list for a survey system: *Register users, Create Survey, Share Survey, Fill Survey, Analyze Surveys* [20]. We won't implement registration or login here, focusing on survey creation and response. The main classes deduced are **Survey** (with properties like title and a collection of questions), **Question** (question text, options), and user roles (Admin and Respondent).

Mapping Actions to Classes:

- **Admin** – actions: *create a survey, add questions, distribute survey*. Methods might be `createSurvey(title)`, `addQuestion(survey, questionText)`, `publishSurvey(survey)`.
- **Respondent (User)** – actions: *take survey, submit answers*. Methods might include `fillSurvey(Survey)` or `answerQuestion(survey, question, answer)`.
- **Manager/Analyst** – action: *analyze survey results*. Could be a method like `analyzeResponses(survey)` (but implementation of analysis is beyond basic scope; we might omit detailed analysis logic).
- **Survey** – actions: *record a response*. Perhaps a method `recordResponse(User, answers)` to store answers, or at least store that a particular user has taken it.

For our example, we will implement Admin and Respondent roles (using separate classes for simplicity) and the Survey/Question classes. We will simulate the actions of adding questions and filling a survey. (We'll skip a detailed Result storage class to keep it simple, but one could imagine a `Response` class holding a respondent's answers).

Example Classes (Java):

```
// Question.java
class Question {
    private String text;
    public Question(String text) {
        this.text = text;
    }
    public String getText() { return text; }
}
```

```java
// Survey.java
import java.util.ArrayList;
import java.util.List;

class Survey {
    private String title;
    private List<Question> questions = new ArrayList<>();

    public Survey(String title) {
        this.title = title;
    }
    public String getTitle() { return title; }

    public void addQuestion(String questionText) {
        questions.add(new Question(questionText));
    }

    public List<Question> getQuestions() {
        return questions;
    }

    // A method to record responses (simplified: just prints or counts
responses)
    public void recordResponse(User respondent, List<String> answers) {
        System.out.println(respondent.getName() + " submitted answers for
survey \"" + title + "\".");
        // In a real app, we'd store the answers correspondingly. Here we'll
just acknowledge.
    }
}
```

```java
// Admin.java (Survey creator)
class Admin extends User {
    public Admin(String name, String email) {
        super(name, email);
    }
    public Survey createSurvey(String title) {
        System.out.println(name + " created a new survey: " + title);
        return new Survey(title);
    }
    public void addQuestionToSurvey(Survey survey, String questionText) {
        survey.addQuestion(questionText);
        System.out.println("Added question: \"" + questionText + "\" to
survey \"" + survey.getTitle() + "\"");
    }
    public void shareSurvey(Survey survey) {
        // In a real system, this might send the survey to users.
        System.out.println(name + " shared the survey \"" +
survey.getTitle() + "\" with respondents.");
```

```
    }
}
```

```
// Respondent.java (Survey participant)
class Respondent extends User {
    public Respondent(String name, String email) {
        super(name, email);
    }
    public void fillSurvey(Survey survey, List<String> answers) {
        // Simulate answering each question (here we just print answers
provided)
        List<Question> questions = survey.getQuestions();
        System.out.println(name + " is filling survey: " +
survey.getTitle());
        for (int i = 0; i < questions.size() && i < answers.size(); i++) {
            System.out.println(" Q: " + questions.get(i).getText());
            System.out.println(" A: " + answers.get(i));
        }
        // Record the response via survey's method
        survey.recordResponse(this, answers);
    }
}
```

```
// (Reuse User.java from the previous example, serving as a base class for
Admin/Respondent)
```

In this design: - **Survey** contains a list of `Question` objects. It provides `addQuestion()` to append questions and a `recordResponse()` method to handle survey submissions (here it simply prints a confirmation). - **Question** holds the text of a question (for simplicity, no options or answer type in this example). - **Admin** (extends `User`) can create a survey, add questions, and share the survey. These methods correspond to the admin's actions: creating and preparing a survey for distribution. - **Respondent** (extends `User`) has a method `fillSurvey` which takes a Survey and a list of answers. It prints each question with the provided answer (simulating the process of a user answering questions) and then calls `survey.recordResponse(this, answers)` – demonstrating the interaction where the Respondent uses the Survey's method to record their participation. In a more complete system, `recordResponse` might save answers to a data structure or database.

For example usage:

```
Admin admin = new Admin("AdminUser", "admin@survey.com");
Survey survey = admin.createSurvey("Customer Satisfaction Survey");
admin.addQuestionToSurvey(survey, "How do you rate our service?");
admin.addQuestionToSurvey(survey, "Would you recommend us to others?");
admin.shareSurvey(survey);

Respondent rachel = new Respondent("Rachel", "rachel@example.com");
List<String> rachelAnswers = new ArrayList<>();
```

```
rachelAnswers.add("Excellent");
rachelAnswers.add("Yes, definitely");
rachel.fillSurvey(survey, rachelAnswers);
```

Output from this sequence might be:

```
AdminUser created a new survey: Customer Satisfaction Survey
Added question: "How do you rate our service?" to survey "Customer
Satisfaction Survey"
Added question: "Would you recommend us to others?" to survey "Customer
Satisfaction Survey"
AdminUser shared the survey "Customer Satisfaction Survey" with
respondents.

Rachel is filling survey: Customer Satisfaction Survey
 Q: How do you rate our service?
 A: Excellent
 Q: Would you recommend us to others?
 A: Yes, definitely
Rachel submitted answers for survey "Customer Satisfaction Survey".
```

This demonstrates the full lifecycle: the Admin creates a survey and adds questions, shares it (conceptually), and the Respondent fills it out. The classes correspond to real-world roles (Admin, Respondent) and entities (Survey, Question), with methods capturing the actions each role performs.

Conclusion: In all these examples, we followed a consistent approach to object-oriented design: 1. **Identify user personas and system entities (nouns)** – these became our classes (e.g., Customer, Account, Product, Patient, Doctor, Complaint, Survey, Question, etc.). 2. **Identify actions (verbs)** each role or entity performs – these became methods (e.g., deposit, withdraw, addProduct, scheduleAppointment, resolveComplaint, createSurvey, fillSurvey). 3. **Identify data (properties)** needed to describe each class – these became class fields (e.g., balance, price, specialization, status, question text).

By mapping verbs to methods and nouns to classes/properties [1] [21], we create a design where each class encapsulates its state and behavior relevant to a real-world concept. The examples also show interactions: one object calling another's methods, which is how complex behavior emerges from simpler components (e.g., a Bank calling Account methods, a Staff calling Complaint methods, a Respondent calling Survey methods). This mirrors real use-case scenarios and keeps the code organized by responsibility. Such an approach leads to modular and extensible code, as new actions or entities can be added by extending classes or adding new methods in the appropriate places. By focusing on the real-world roles and actions, beginners can more easily design classes that make sense and fulfill the software requirements in an object-oriented way.

---

[1] [3] Programming in Nouns and Verbs - Clear Launch
https://www.clearlaunch.com/programming-nouns-verbs/

2 4 6 21 oop - Techniques For Identifying Classes And Their Responsibilites - Stack Overflow
https://stackoverflow.com/questions/15551584/techniques-for-identifying-classes-and-their-responsibilites

5 9 10 Understanding Class Diagram with Example
https://boardmix.com/articles/class-diagram/

7 8 UML Class diagram for online shopping: Step-by-step tutorial | Gleek | Gleek
https://www.gleek.io/blog/class-online-shopping

11 5 Free Hospital Management Class Diagrams
https://edrawmax.wondershare.com/class-diagram/for-hospital-management.html

12 13 14 15 16 Microsoft Word - 10.FJ23C322
https://foundryjournal.net/wp-content/uploads/2024/04/10.FJ23C322.pdf

17 18 19 20 Simple Survey Application - CodeProject
https://www.codeproject.com/Articles/667426/Simple-Survey-Application