Task A:

Scenario: Library Management System

In our scenario, we have a Library Management System where users can check out books. Initially, the code for checking out a book is straightforward but as requirements evolve, more conditions are added, making the code hard to understand and prone to changes.

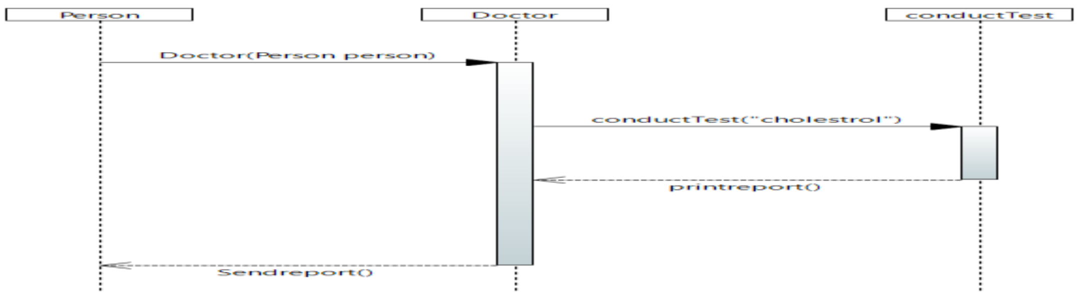
Without Encapsulation:

```
public class Library {
    public void checkoutBook(Customer customer, Book book) {
        if (customer != null && customer.getFine() <= 0.0 &&
            customer.getCard() != null && customer.getCard().getExpiration() == null &&
            book != null && !book.isCheckedOut()) {
            customer.addBook(book);
            book.setCheckedOut(true);
        }
    }
}
```

With Encapsulation:

```
public class Library {
    public void checkoutBook(Customer customer, Book book) {
        if (customer.canCheckout(book)) {
            customer.checkout(book);
        }
    }
}


public class Customer {
    private double fine;
    private Card card;
    private List<Book> books;
    public boolean canCheckout(Book book) {
        return fine <= 0.0 && card != null && card.getExpiration() == null && book != null &&
!book.isCheckedOut();
    }

    public void checkout(Book book) {
        books.add(book);
        book.setCheckedOut(true);
    }
}
```

Interaction Diagram:



Task B: Abstract Factory Pattern

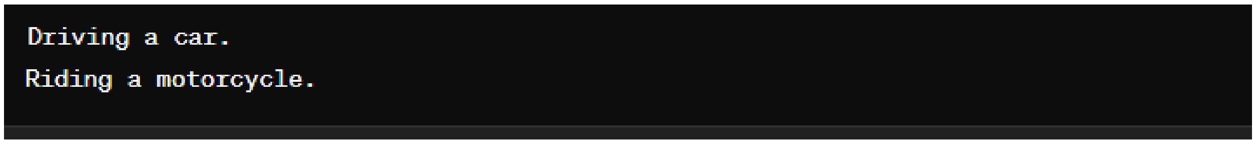
Scenario: Vehicle Manufacturing System

In our scenario, we have a Vehicle Manufacturing System where we produce different types of vehicles such as cars and motorcycles. We want to implement an Abstract Factory pattern to allow the creation of families of related objects without specifying their concrete classes.

```
interface Vehicle {
    void drive();
}
```

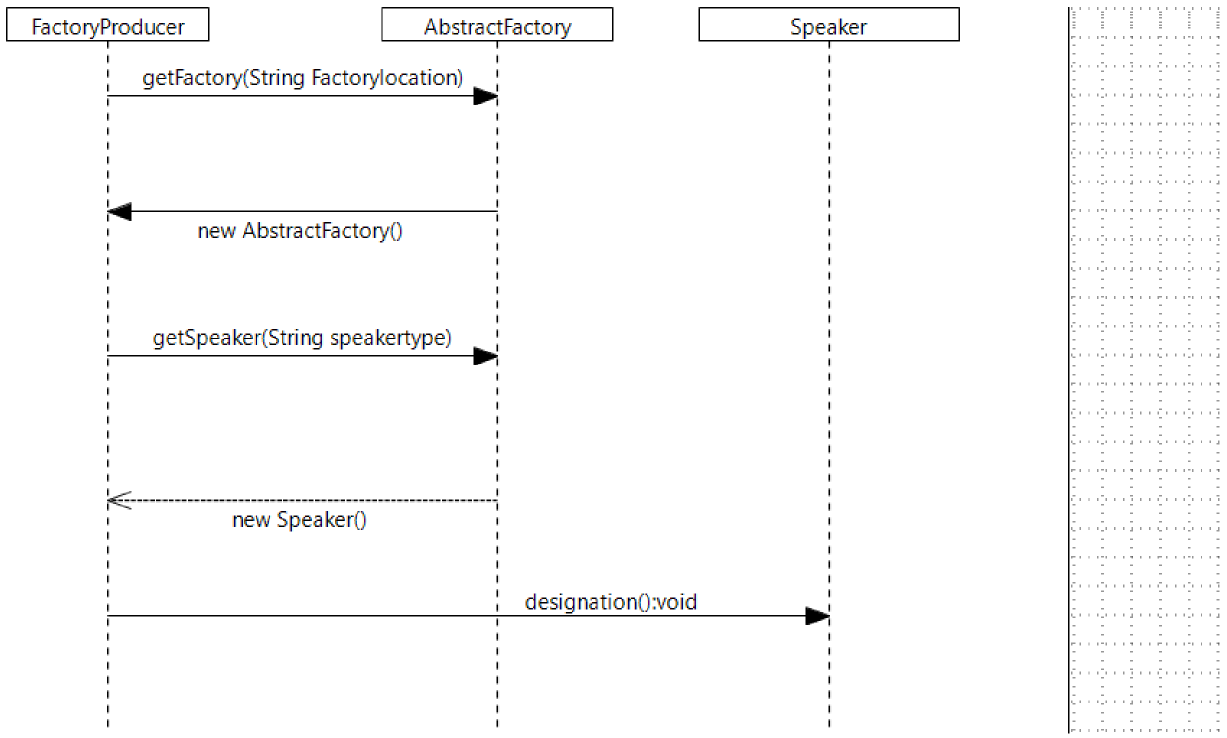
```java
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car.");
    }
}

class Motorcycle implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a motorcycle.");
    }
}

interface VehicleFactory {
    Vehicle createVehicle();
}

class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}

class MotorcycleFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Motorcycle();
    }
}

public class Main {
    public static void main(String[] args) {
        VehicleFactory carFactory = new CarFactory();
        Vehicle car = carFactory.createVehicle();
        car.drive();

        VehicleFactory motorcycleFactory = new MotorcycleFactory();
        Vehicle motorcycle = motorcycleFactory.createVehicle();
        motorcycle.drive();
    }
}
```

Output:



```
Driving a car.
Riding a motorcycle.
```

Interaction Diagram:


FactoryProducer
AbstractFactory
Speaker
getFactory(String Factorylocation)
new AbstractFactory()
getSpeaker(String speakertype)
new Speaker()
designation():void