Issue1:
Current way of creating different models of planes is inextensible because if we add a new attribute for a plane then the same attribute should be updated in all the conditions of the CASE statement.

Resolution:

1. We can use Factory pattern in order to create different types of planes. With the factory pattern, we can encapsulate the logic of object creation and it'll make it easy to add new types of objects in the future without requiring changes to the client code.

```
public class PlaneFactory {

    public PassengerPlane createPlane(String model) {
        switch (model) {
            case "A380":
                return new A380();
            case "A350":
                return new A350();
            case "Embraer 190":
                return new Embraer190();
            case "Antonov AN2":
                return new AntonovAN2();
            default:
                throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}
```

2. Generally, we use an interface to define object behavior and implement that interface for object creation while setting respective attributes. But, in this case we've used super class PassengerPlane with all the common attributes.
3. Constructor of super class will set the default values for all objects of PassengerPlane.

```
public class PassengerPlane {

    public String model;
    public int passengerCapacity;
    public int crewCapacity;
    public boolean air_hostesses_required;

    public PassengerPlane(String model) {
        this.model = model;
        this.air_hostesses_required = true;
    }
}
```

4. To add a new type of plane, we just need to create a new subclass of PassengerPlane and add a case to the factory's switch statement. In this way, we can make the object creation more extensible.

```
public class A380 extends PassengerPlane {
    public A380() {
        super("A380");
        this.passengerCapacity = 500;
        this.crewCapacity = 42;
    }
}

public class A350 extends PassengerPlane {
    public A350() {
        super("A350");
        this.passengerCapacity = 320;
        this.crewCapacity = 40;
    }
}

public class Embraer190 extends PassengerPlane {
    public Embraer190() {
        super("Embraer 190");
        this.passengerCapacity = 25;
        this.crewCapacity = 5;
    }
}

public class AntonovAN2 extends PassengerPlane {
    public AntonovAN2() {
        super("Antonov AN2");
        this.passengerCapacity = 15;
        this.crewCapacity = 3;
    }
}
```

Issue2:
Currently, there is no provision to notify the passengers about the flight updates like change in schedule, cancellations etc.

Resolution:
We can leverage the observer pattern in order to notify all the passengers of a flight.

1. Create an interface that defines the methods that the observers can use to register themselves, unregister themselves, and receive updates from the subject.

```
public interface FlightNotifications {
   public void addPassenger(Passenger passenger);
   public void removePassenger(Passenger passenger);
   public void notifyPassengers();
}
```

2. The existing class 'ScheduledFlight' which has a list of passengers can implement this interface to add, remove and notify the passengers. Following is the implementation of those methods:

```
   public void addPassenger(Passenger passenger) {
       this.passengers.add(passenger);
   }

   public void removePassenger(Passenger passenger) {
       this.passengers.remove(passenger);
   }

   public void notifyPassengers() {
       for (Passenger passenger : this.passengers) {
           passenger.update(this);
       }
   }

   public void setDepartureTime(Date departureTime) {
       this.departureTime = departureTime;
       this.notifyPassengers();
   }
```

3. Now, we need to have an interface for receiving the notification in Passenger. For that, we will create an interface for observer which will be implemented in Passenger.

```
public interface FlightNotificationObserver {
   public void update(ScheduledFlight flight);
}
```

4. This behavior will be implemented in Passenger in the following way:

```
public void update(ScheduledFlight flight) {
       System.out.println("Notification for user " + this.name + ":
Flight " + flight.getFlightNumber() + " has changed Departure Time to " +
flight.getDepartureTime());
   }
```

5. With this, we can achieve the notification functionality to passengers upon change in schedule of a flight.