interface Vehicle {
int set_num_of_wheels()
int set_num_of_passengers()
boolean has_gas()
}

a) Explain how you can use the pattern to create car and plane class?
b) Use a different design pattern for this solution.

Ans to the ques no a :

Here, Car and Plane will implement Vehicle and override three methods. This pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor.*Vehicle* interface which will be implemented by several concrete classes. A Vehicle*Factory* will be used to fetch objects from this family:

```
public interface Vehicle {
  int set_num_of_wheels();
  int set_num_of_passengers();
  boolean has_gas();
}
```

```
public class Car implements Vehicle {
  private int wheel, passengers;
  private boolean has_gas;
  public Car(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public int set_num_of_wheels() {
    return this.wheel;
  }
  @Override
  public int set_num_of_passengers() {
    return this.passengers;
  }
  @Override
  public boolean has_gas() {
    return this.has_gas;
  }
}
```

```
public class Plane implements Vehicle {
  private int wheel, passengers;
```

```
  private boolean has_gas;
  public Plane(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public int set_num_of_wheels() {
    return this.wheel;
  }
  @Override
  public int set_num_of_passengers() {
    return this.passengers;
  }
  @Override
  public boolean has_gas() {
    return this.has_gas;
  }
}
```

Now we can create a factory that takes the vehicle type,number of wheels, number of passengers and has_gas as an argument and returns the appropriate implementation of this interface:

```
public class VehicleFactory {
  public static Vehicle getVehicle(String vehicleType, int wheel,
int passengers, boolean has_gas){
    if(vehicleType.equalsIgnoreCase("car")){
      return new Car(wheel,passengers,has_gas);
    }
    else if(vehicleType.equalsIgnoreCase("plane")){
      return new Plane(wheel,passengers,has_gas);
    }
    return null;
  }
}
```

When the implementation of an interface or an abstract class is expected to change frequently and when the current implementation cannot comfortably accommodate new change then we use factory design pattern.

Ans to the ques no b :

Here I use Abstract Factory Design Pattern for this solution. Abstract Factory design pattern is one of the Creational patterns. Abstract Factory pattern is almost similar to Factory Pattern except the fact that its more like factory of factories. In the Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class. Then an Abstract Factory class that will return the sub-class based on the input factory class. At first, it seems

confusing but once you see the implementation, it’s really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Here is the super class and subclasses :

```
package two_b;
public interface Vehicle {
  int set_num_of_wheels();
  int set_num_of_passengers();
  boolean has_gas();
}
```

```
package two_b;
public class Car implements Vehicle {
  private int wheel, passengers;
  private boolean has_gas;
  public Car(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public int set_num_of_wheels() {
    return this.wheel;
  }
  @Override
  public int set_num_of_passengers() {
    return this.passengers;
  }
  @Override
  public boolean has_gas() {
    return this.has_gas;
  }
}
```

```
package two_b;
public class Plane implements Vehicle {
  private int wheel, passengers;
  private boolean has_gas;
  public Plane(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public int set_num_of_wheels() {
    return this.wheel;
  }
  @Override
  public int set_num_of_passengers() {
    return this.passengers;
  }
```

```
  @Override
  public boolean has_gas() {
    return this.has_gas;
  }
}
```

First of all we need to create a Abstract Factory interface or abstract class.

```
package two_b;
public interface AbstractFactory{
  public Vehicle createVehicle();
}
```

Notice that, `createVehicle()` method is returning an instance of super class Vehicle. Now our factory classes will implement this interface and return their respective sub-class.

```
package two_b;
public class CarFactory implements AbstractFactory{
  private int wheel, passengers;
  private boolean has_gas;
  public CarFactory(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public Vehicle createVehicle() {
    return new Car(wheel,passengers,has_gas);
  }
}
```

```
package two_b;
public class PlaneFactory implements AbstractFactory {
  private int wheel, passengers;
  private boolean has_gas;
  public PlaneFactory(int wheel, int passengers, boolean has_gas) {
    this.wheel = wheel;
    this.passengers = passengers;
    this.has_gas = has_gas;
  }
  @Override
  public Vehicle createVehicle() {
    return new Plane(wheel,passengers,has_gas);
  }
}
```

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

```
package two_b;
public class VehicleFactory {
  public static Vehicle getVehicle(AbstractFactory factory){
    return factory.createVehicle();
  }
}
```

its a simple class and `getVehicle` method is accepting `AbstractFactory` argument and returning Vehicle object. At this point the implementation must be getting clear.

Let’s write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

```
package two_b;
public class AbstractFactoryPatternMain {
  public static void main(String[] args) {
    testAbstractFactory();
  }
  private static void testAbstractFactory() {
    Vehicle car = VehicleFactory.getVehicle(new
CarFactory(4,5,true));
    Vehicle plane = VehicleFactory.getVehicle(new
PlaneFactory(10,25,false));
  }
}
```