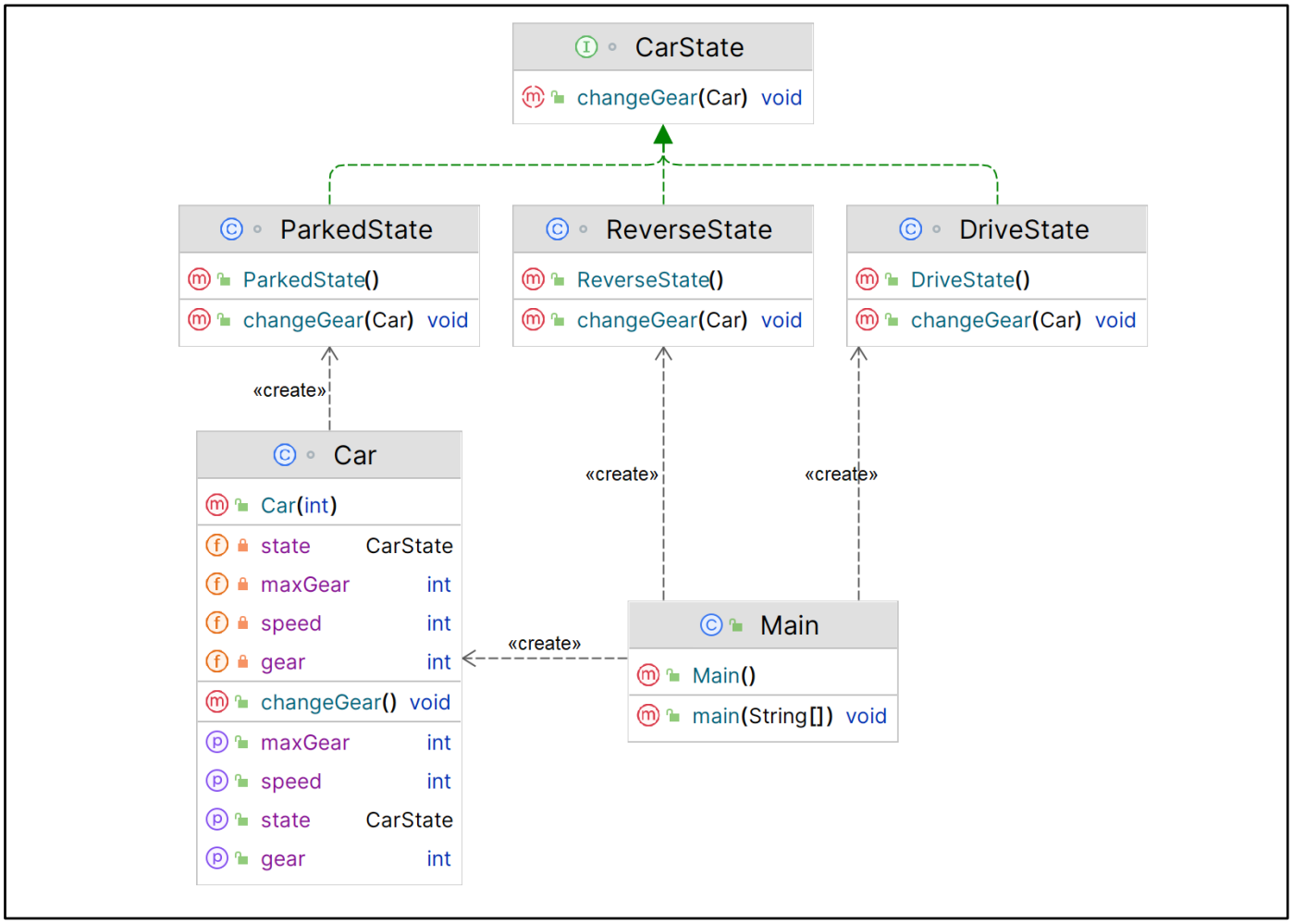
Assignment 11: State Design Pattern

What is State Design Pattern?

State is a behavioural design pattern that lets an object alter its behaviour when its internal state changes. It appears as if the object changed its class.

Structure (Class Diagram)



Implementation (Code)

```
// Interface for CarState
interface CarState {
    void changeGear(Car car);
}

// Concrete class for ParkedState
class ParkedState implements CarState {

    public void changeGear(Car car) {
        // Car can only change gear when it's not parked
        System.out.println("Cannot change gear while car is parked.");
```

```
    }
}

// Concrete class for DriveState
class DriveState implements CarState {
      public void changeGear(Car car) {
       // Car can change gear to higher gear when driving at certain speed
       if (car.getSpeed() < 20) {
          System.out.println("Cannot change to higher gear when car is moving slowly.");
       } else if (car.getGear() >= car.getMaxGear()) {
          System.out.println("Cannot shift to higher gear, already in top gear.");
       } else {
          car.setGear(car.getGear() + 1);
          System.out.println("Changed gear to " + car.getGear());
       }
    }
}

// Concrete class for ReverseState
class ReverseState implements CarState {

    public void changeGear(Car car) {
       // Car can only change to reverse gear when speed is 0
       if (car.getSpeed() > 0) {
          System.out.println("Cannot shift to reverse gear when car is moving forward.");
       } else {
          car.setGear(-1);
          System.out.println("Changed gear to reverse");
       }
    }
}

// Context class for Car
class Car {
    private int speed;
    private int gear;
    private int maxGear;
    private CarState state;

    public Car(int maxGear) {
       this.speed = 0;
       this.gear = 0;
       this.maxGear = maxGear;
       this.state = new ParkedState();
    }
```

```
    public void changeGear() {
        this.state.changeGear(this);
    }

    // Getters and setters for speed, gear, and maxGear

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return this.speed;
    }

    public void setGear(int gear) {
        this.gear = gear;
    }

    public int getGear() {
        return this.gear;
    }

    public int getMaxGear() {
        return this.maxGear;
    }

    // Method to set the state of the car
    public void setState(CarState state) {
        this.state = state;
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Car car = new Car(4);

        // Car starts in parked state
        car.changeGear(); // Output: "Cannot change gear while car is parked."

        // Car can shift to reverse gear when speed is 0
        car.setState(new ReverseState());
        car.changeGear(); // Output: "Changed gear to reverse"
```

```
        // Car cannot shift to higher gear when moving slowly
        car.setState(new DriveState());
        car.setSpeed(10);
        car.changeGear(); // Output: "Cannot change to higher gear when car is moving slowly."

        // Car can shift to higher gear when moving at certain speed
        car.setSpeed(25);
        car.changeGear(); // Output: "Changed gear to 0"
        car.changeGear(); // Output: "Changed gear to 1"
        car.changeGear(); // Output: "Changed gear to 2"
        car.changeGear(); // Output: "Changed gear to 3"
        car.changeGear(); // Output: "Changed gear to 4"

        // Car cannot shift to higher gear when already in top gear
        car.changeGear(); // Output: "Cannot shift to higher gear, already in top gear"
    }
}
```

Output

```
Cannot change gear while car is parked.
Changed gear to reverse
Cannot change to higher gear when car is moving slowly.
Changed gear to 0
Changed gear to 1
Changed gear to 2
Changed gear to 3
Changed gear to 4
Cannot shift to higher gear, already in top gear.
```

Applicability

1. Use the **State pattern** when you have an object that **behaves differently** depending on its current state, the number of states is enormous, and the **state-specific code** changes frequently.
2. Use the pattern when you have a class polluted with massive conditionals that alter how the class **behaves** according to the current values of the class's fields.
3. Use State when you have a lot of **duplicate code** across similar states and transitions of a condition-based **state machine**.