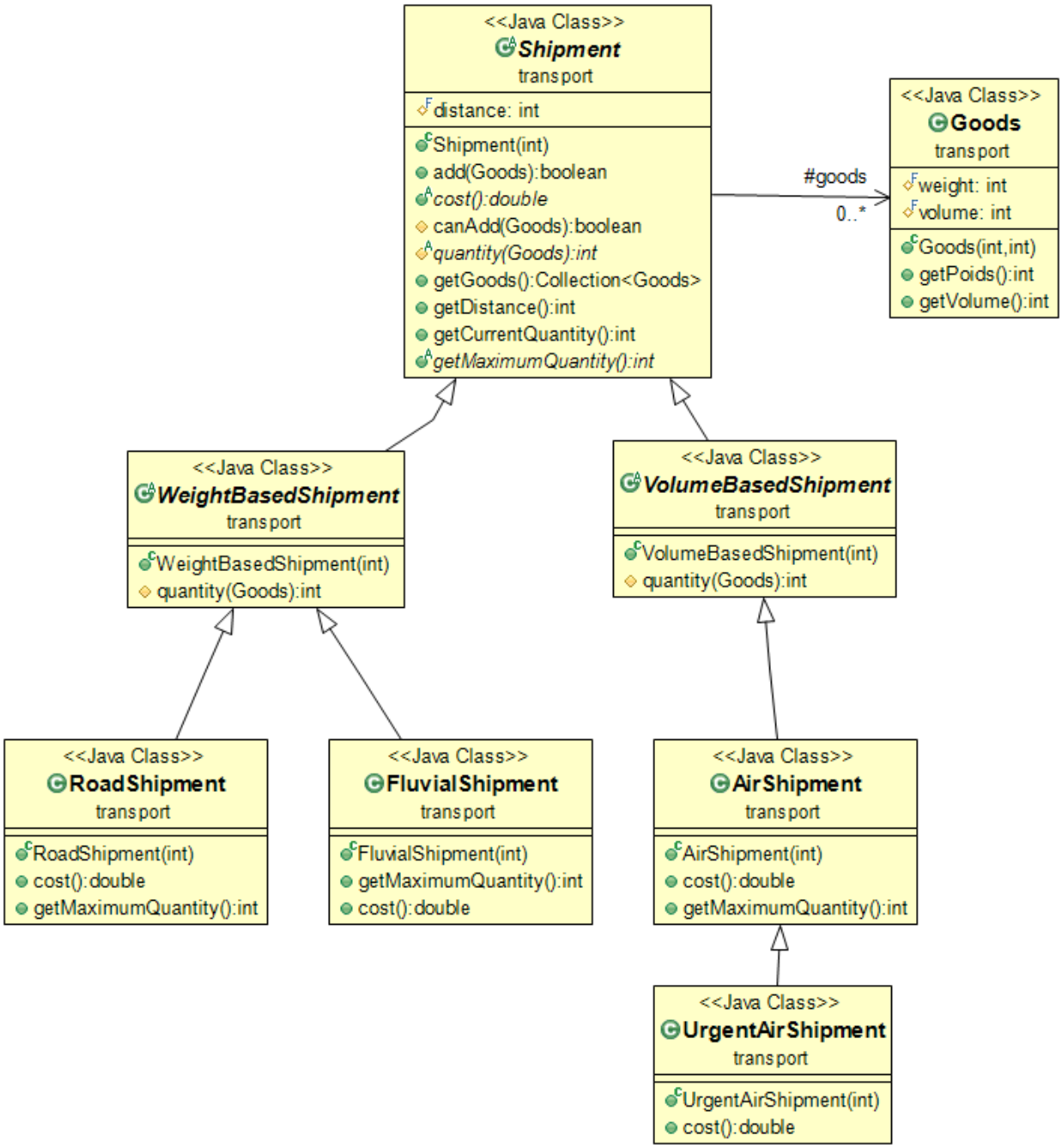
Transports

Q2 et Q3

```
    // Weight
    public void add(Goods goods) throws IllegalStateException {
        if (this.getCurrentWeight() + this.goods.getWeight() > this.getMaximumWeight()) {
            throw new IllegalStateException("goods cannot be added");
        }
        this.currentWeight = this.getCurrentWeight() + this.goods.getWeight();
        this.goods.add(goods);
    }

    // Volume
    public void add(Goods goods) throws IllegalStateException {
        if (this.getCurrentVolume() + this.goods.getVolume() > this.getMaximumVolume()) {
            throw new IllegalStateException("goods cannot be added");
        }
        this.currentVolume = this.currentVolume + this.goods.getVolume();
        this.goods.add(goods);
    }

    // Code identique “au paramètre” près (poids ou volume). Les 2 étant de type entier,
    // on peut proposer d’utiliser un attribut commun (quantity) du même type.


public abstract class Shipment {

    protected Collection<Goods> goods = new HashSet<Goods>();
    protected final int distance;
    protected int currentQuantity;

    public Shipment(int distance) {
        this.distance = distance;
        this.currentQuantity = 0;
    }

    public void add(Goods goods) throws IllegalStateException {
        if (this.getCurrentQuantity() + this.quantity(goods) > this.getMaximumQuantity()) {
            throw new IllegalStateException("goods cannot be added");
        }
        this.currentQuantity = this.currentQuantity + this.quantity(goods);
        this.goods.add(goods);
    }

    public abstract double cost(); // Spécifique à chaque Shipment

    protected abstract int quantity(Goods goods); // Weight ou Volume selon le cas

    public abstract int getMaximumQuantity();

    public Collection<Goods> getGoods() {
        return this.goods;
    }

    public int getDistance() {
        return this.distance;
    }

    public int getCurrentQuantity() {
        return this.currentQuantity;
    }
}
```


<<Java Class>>
Shipment
transport
distance: int
Shipment(int)
add(Goods):boolean
cost():double
canAdd(Goods):boolean
quantity(Goods):int
getGoods():Collection<Goods>
getDistance():int
getCurrentQuantity():int
getMaximumQuantity():int
#goods
0..*
<<Java Class>>
Goods
transport
weight: int
volume: int
Goods(int,int)
getPoids():int
getVolume():int
<<Java Class>>
WeightBasedShipment
transport
WeightBasedShipment(int)
quantity(Goods):int
<<Java Class>>
VolumeBasedShipment
transport
VolumeBasedShipment(int)
quantity(Goods):int
<<Java Class>>
RoadShipment
transport
RoadShipment(int)
cost():double
getMaximumQuantity():int
<<Java Class>>
FluvialShipment
transport
FluvialShipment(int)
getMaximumQuantity():int
cost():double
<<Java Class>>
AirShipment
transport
AirShipment(int)
cost():double
getMaximumQuantity():int
<<Java Class>>
UrgentAirShipment
transport
UrgentAirShipment(int)
cost():double

Q4

```
public abstract class ShipmentTest {

	protected abstract Shipment createShipment();
	protected abstract Goods tooBigGoods();
	protected abstract double getExpectedCostValue();

	protected Goods okGoods() {
		return new Goods(100,100);
	}

	protected Shipment shipment;
	protected Goods okGoods;


	@Before
	public void setupBefore() {
		this.shipment = this.createShipment();
		this.okGoods = this.okGoods();
	}

	@Test // Q4.1
	public void testAddWhenOk() {
		assertEquals(0, shipment.getCurrentQuantity());
		shipment.add(this.okGoods);
		assertTrue(shipment.getGoods().contains(okGoods));
		assertEquals(shipment.quantity(okGoods), shipment.getCurrentQuantity());
	}

	@Test(expected=IllegalStateException.class) // Q4.2
	public void testAddThrowsExceptionWhenQuantityTooBig() {
		try {
			shipment.add(this.okGoods);
		} catch(IllegalStateException e) {
			// exception must not be thrown in this case
			fail();
		}
		shipment.add(this.tooBigGoods());
	}

	@Test // Q4.3
	public void testCostWhenAddGoodsIsOk() {
		shipment.add(this.okGoods);
		assertEquals(this.getExpectedCostValue(), shipment.cost(), 0.00001);
	}

	@Test // Q4.4
	public void testCostDoesNotChangeWhenAddGoodsIsNotOk() {
		shipment.add(this.okGoods);
		double cost = shipment.cost();
		try {
			shipment.add(this.tooBigGoods());
			// add should not be done
			fail();
		} catch(Exception e) {
			// do nothing : add has not been done
		}
		assertEquals(cost,shipment.cost(),0.00001);
	}
```

}

```
public class RoadShipmentTest extends ShipmentTest {

    @Override
    protected Shipment createShipment() {
        return new RoadShipment(10);
    }

    @Override
    protected Goods tooBigGoods() {
        return new Goods(RoadShipment.MAX_QTY + 1, 100);
    }

    @Override
    protected double getExpectedCostValue() {
        return 4000;
    }
}
```

```
public class FluvialShipmentTest extends ShipmentTest {

    @Override
    protected Shipment createShipment() {
        return new FluvialShipment(10);
    }

    @Override
    protected Goods tooBigGoods() {
        return new Goods(FluvialShipment.MAX_QTY + 1, 100);
    }

    @Override
    protected double getExpectedCostValue() {
        return 100;
    }
}
```

```
public class AirShipmentTest extends ShipmentTest {

    @Override
    protected Shipment createShipment() {
        return new AirShipment(10);
    }

    @Override
    protected Goods tooBigGoods() {
        return new Goods(100, AirShipment.MAX_QTY + 1);
    }

    @Override
    protected double getExpectedCostValue() {
        return 500;
    }
}
```

```
public class UrgentAirShipmentTest extends AirShipmentTest {

    @Override
    protected Shipment createShipment() {
        return new UrgentAirShipment(10);
    }

    @Override
    protected double getExpectedCostValue() {
        return super.getExpectedCostValue()*2;
    }
}
```