A Train is comprised of a Locomotive followed by one or more Cars. Locomotive and Car are subclasses of Train. For example, Car c1 is first coupled to Locomotive l by l.couple(c1), then c2 is next coupled to c1, etc. Later, two cars can be separated, e.g., car c2 can be decoupled from c1 by c2.decouple() which breaks the connection between c1 and c2 (but c3 is still coupled to c2).

There are several types of Visitors to Trains. A Train is itself one type of Visitor. When Train t1 visits train t2, the first car of t2 is coupled to the end of t1, making t1 longer. The above train, for example, is created by:

```
Locomotive l  = new Locomotive();
Car       c1 = new Car();
Car       c2 = new Car();
Car       c3 = new Car();

   c1.acceptVisitor(l);  // c1 is coupled to the locomotive
   c2.acceptVisitor(l);  // c2 is coupled to c1 (which is coupled to l)
   c3.acceptVisitor(l);  // and finally c3 is added to the end
```

Regarding the Visitor Pattern, we do not distinguish between Locomotive versus Car as "visitee" (only the class Train is visited), but there would be many other Visitors to Train (passengers, engineers, etc., as well as Train), therefore Visitor must be an interface. First write Visitor below, then finish writing Train on the next page (don’t worry about its subclasses Locomotive and Car).

1. [10%] Write the interface Visitor:

   public interface Visitor {
2. [35%] Write the class Train:

```java
public class Train
{
    protected Train prev; // this is the link towards the front of the train
    protected Train next; // this is the link towards the back of the train

    public Train()
    {
    }

    public void couple(Train t) {
    }

    public void decouple() {
        if (prev == null)
            return;
        prev.setNext(null);
        prev = null;
    }

    public Train getPrev()
    {
        return prev;
    }

    public Train getNext()
    {
        return next;
    }

    public void setPrev(Train t)
    {
        prev = t;
    }

    public void setNext(Train t)
    {
        next = t;
    }

    public boolean hasPrev()
    {
        return prev != null;
    }

    public boolean hasNext()
    {
        return next != null;
    }

    public void acceptVisitor()
    {
        // when visiting another Train, couple that Train to the end of this Train
        // using some of the above methods and recursion. No while or for loops:
    }

    public void visit(Train t) {
    }
```
A toy railroad consists of a binary tree of branches. Each Branch has a length of track at the end up to two Branches (a left and a right) might connect. For instance, b1 is 5 units long and has Branches b2 and b3. Branch b2 is 4 units long and b3 is only 2 units long. Note that b3 has only one sub-Branch, b6.

```java
Branch b1 = new Branch(5);
Branch b2 = new Branch(4);
Branch b3 = new Branch(2);
b1.setLeft(b2);
b1.setRight(b3);
... and etc. for b4, b5, and b6 to build the whole binary tree of branches.
```

A Branch is either Operational or Defunct (i.e., “no longer working”). It is initially Operational. A Train can only travel down an Operational Branch. If b2 were Defunct, the only path a train could travel is b1 to b3 to b6, even if b4 and b5 were still Operational.

The method `public int getOperationalLength()` would return the total length of all sub-branches that could be traveled from b1 down.

```java
b1.getOperationalLength();  // returns 5 + 4 + 2 + etc.
b4.getOperationalLength();  // returns only length of b4 (no sub-branches)
b2.setDefunct();            // with b2 defunct, trains cannot get to b4 or b5
b2.getOperationalLength();  // returns zero because b2 is now Defunct
b1.getOperationalLength();  // returns only length of b1 and b3 and b6
```

On the next page, finish writing Branch, including the method `public int getOperationalLength()`. Use the State Pattern to minimize the use of conditionals. Write Branch, State, and the two states Operational and Defunct. Note that the method `getLength()` is state dependent, because it returns zero if the given branch is defunct, and `getOperationalLength()` is tree recursive (the sum of the given length plus that of the sub-branches).
public class Branch {
    public int    length;
    public Branch left;
    public Branch right;
    public State  currentState
    public State  defunct;
    public State  operational;

    public Branch(int length) {
        this.length = length;
        defunct     = new Defunct();
        operational = new Operational();
        setOperational();
    }

    public void setOperational() { currentState = operational; }
    public void setDefunct()     { currentState = defunct; }

    public void setLeft (Branch b) { left  = b; }
    public void setRight(Branch b) { right = b; }

    public Branch getLeft()  { return left;  }
    public Branch getRight() { return right; }

    // write public int getLength() and public int getOperationalLength() below:
4. [20%] Describe the State Pattern and Visitor Pattern in the context of the above Train and Branch applications. Explain the design goals of each pattern separately, and how it helps the architecture. Also explain any drawbacks or limitations or negative issues with their use.