This quiz has elements of recursive data structures, the Visitor Pattern and the State Pattern. A Bush is in one of three disjoint states: Dormant, Growing, or Bearing. The Bush becomes Dormant after a FallV visitor. When SpringV visits it becomes Growing. A SummerV brings the Bearing state.

A Bush is either a Stem, Fork, Bud or Flower. A GrowV visitor uses the method grow() to make all Stems longer and to replace each Bud with a new Fork (when in the Growing state) or a new Flower (when in the Blooming state). In Steps 1-4, below, a SpringV induces the Growing state (1), then a GrowV causes growth (2) and again (3). After a SummerV visit, the Bush is in the Bearing state, and growV then produces new Flowers (4).

In the following, use 'pure' visitor and state patterns. Do NOT use instanceof or conditionals (if, while, the '?-:' expression, etc.). Note: Bush has ONE state-dependent method, namely grow(). Each Stem gets longer and each Bud is replaced by a Fork or Flower (depending on state). Flowers do not grow in this simplification.
1) [15%] Use these visitors to complete the sequence 1-4 above.

```java
SpringV springV = new SpringV(); // Bush in Growing state
SummerV summerV = new SummerV(); // Bush in Bearing state
GrowV growV = new GrowV(); // causes recursive growth
Bush george = new Stem();

george.acceptVisitor(springV); // in Growing state (step 1)
```

```java
// now make george go to steps 2, 3, and 4.
george.acceptVisitor(growV); // does step 2 (new Fork)
george.acceptVisitor(growV); // does step 3 (new Forks)
george.acceptVisitor(summerV); // now in Bearing state
george.acceptVisitor(growV); // so growth forms Flowers.
```

Next, to get ready for question 2, let's look at the code. Here are the completed Bush and Fork classes for you to use:

```java
abstract public class Bush {
    private Growing growing = new Growing();
    private Dormant dormant = new Dormant();
    private Bearing bearing = new Bearing();
    private State state = growing;

    public void setGrowing() { state = growing; }
    public void setDormant() { state = dormant; }
    public void setBearing() { state = bearing; }

    abstract public void acceptVisitor(Visitor v);
    abstract public void grow();
}
```

```java
public class Fork extends Bush {
    private Stem left = new Stem();
    private Stem right = new Stem();

    public void acceptVisitor(Visitor v) {
        left.acceptVisitor(v);
        right.acceptVisitor(v);
    }
}
public void grow() {
    left.grow();
    right.grow();
}
The Flower class is also complete. It accepts visitors, but Flowers don’t grow in this simplification.

```java
public class Flower extends Bush {
    public void acceptVisitor(Visitor v) { v.visit(this); }
    public void grow() {}
}
```

The Stem class is also complete.

```java
public class Stem extends Bush {
    private Bush next;
    private int length = 10; // initial length

    public Stem() { next = new Bud(this); } // Bud needs parent

    public void setNext(Bush b) { next = b; }
    public int getLength() { return length; }
    public void setLength(int l) { length = l; }
    public void acceptVisitor(Visitor v) {
        v.visit(this);
        next.acceptVisitor(v);
    }
    public void grow() { state.grow(); }
}
```

2. [15%] This Bud’s for you ... to finish. Remember to use the State Pattern.

```java
public class Bud extends Bush {
    private Stem parent;

    public Bud(Stem s) { parent = s; }
    public void acceptVisitor(Visitor v) { v.visit(this); }
    public Stem getParent() { return parent; }
    public void grow() { state.grow(); }
}
```
Here is the base class State, and Dormant

abstract public class State {
    abstract public void grow(Stem s);
    abstract public void grow(Fork f);
    abstract public void grow(Bud b);
    abstract public void grow(Flower f);
}

public class Dormant extends State { // no growth in this state
    public void grow(Stem s) {}
    public void grow(Fork f) {}
    public void grow(Bud b) {}
    public void grow(Flower f) {}
}

3. [50%] Finish writing Growing and Bearing.

public class Growing extends State {
    public void grow(Stem s) { s.setLength(1.25*s.getLength()); }

    public void grow(Fork f) {}

    public void grow(Bud b) {
        Stem s = b.getParent();
        Fork f = new Fork();
        s.setNext(f);
    }
    public void grow(Flower f) {}
}

public class Bearing extends State {
    public void grow(Stem s) { s.setLength(1.25*s.getLength()); }

    public void grow(Fork f) {}

    public void grow(Bud b) {
        Stem s = b.getParent();
        Flower f = new Flower();
        s.setNext(f);
    }
    public void grow(Flower f) {}
}
And the Visitor hierarchy

    abstract public class Visitor {
        abstract public void visit(Stem s);
        abstract public void visit(Fork f);
        abstract public void visit(Flower f);
        abstract public void visit(Bud b);
    }

    public class WinterV extends Visitor {
        public void visit(Stem s)   { s.setDormant(); }
        public void visit(Fork f)   { f.setDormant(); }
        public void visit(Flower f) { f.setDormant(); }
        public void visit(Bud b)    { b.setDormant(); }
    }

    public class SpringV extends Visitor {
        public void visit(Stem s)   { s.setGrowing(); }
        public void visit(Fork f)   { f.setGrowing(); }
        public void visit(Flower f) { f.setGrowing(); }
        public void visit(Bud b)    { b.setGrowing(); }
    }

    public class SummerV extends Visitor {
        public void visit(Stem s)   { s.setBearing(); }
        public void visit(Fork f)   { f.setBearing(); }
        public void visit(Flower f) { f.setBearing(); }
        public void visit(Bud b)    { b.setBearing(); }
    }

4. [20%] Finish the GrowV. Remember growth occurs to Stems and Buds only, not Flowers, and only recursively to Forks.

    public class GrowV extends Visitor {
        public void visit(Stem s)   { s.grow(); }
        public void visit(Fork f)   {}  
        public void visit(Flower f) {}  
        public void visit(Bud b)    { b.grow(); }
    }