Fruit is an abstract class which implements EdibleI:

    interface EdibleI { String getTaste(); }

Every kind of Fruit has a method public String getColor().

Banana and Strawberry are two kinds of Fruit.

Every Fruit is initially in the Unripe state then can become Ripe or Rotten. The color and taste of any Fruit depends on its state:

<table>
<thead>
<tr>
<th></th>
<th>state</th>
<th>color</th>
<th>taste</th>
</tr>
</thead>
<tbody>
<tr>
<td>for Banana:</td>
<td>Unripe</td>
<td>“green”</td>
<td>“yuck!”</td>
</tr>
<tr>
<td></td>
<td>Ripe</td>
<td>“yellow”</td>
<td>“yum!”</td>
</tr>
<tr>
<td></td>
<td>Rotten</td>
<td>“black”</td>
<td>“urp!”</td>
</tr>
<tr>
<td>for Strawberry:</td>
<td>Unripe</td>
<td>“yellow”</td>
<td>“bleah!”</td>
</tr>
<tr>
<td></td>
<td>Ripe</td>
<td>“red”</td>
<td>“ahhhhh!”</td>
</tr>
<tr>
<td></td>
<td>Rotten</td>
<td>“purple”</td>
<td>“pituii!”</td>
</tr>
</tbody>
</table>

Fruit also implements RipenI:

    interface RipenI {
        void setRipe();
        void setRotten();
    }

For example, f.setRipe() sets a Fruit f to the Ripe state.

A ListC garden contains many instances of Banana and Strawberry.

The Fruit in the garden are visited by Visitors such Sunshine or Frost.

Every day sunbeam (an instance of Sunshine) visits garden and most of the Fruit become Ripe as a result of her repeated visits.

1. [5 points] Complete the line with the “>>” to show one visit to all the Fruit in garden.
Sunshine sunbeam = new Sunshine();
garden.start();
while (garden.more()) {
    Fruit f = (Fruit)garden.get();
}

Later you pick a Fruit f = (Fruit)garden.get(); and appreciate f’s color by calling f.getColor(); and taste it by f.getTaste();

Another kind of Visitor is a Frost, which, if it visits a Fruit, unconditionally sets that Fruit’s state to Rotten.
    Frost jack = new Frost();
    Fruit f    = (Fruit)garden.get();
then, for example, f.getTaste() would return either “urp!” or “pituii!”

2. [30 points] Write the abstract Fruit class and the Banana class and use the State Pattern to achieve state-dependent color and taste. Remember to allow for accepting Visitors. In addition to the classes Fruit and Banana you need to write a state class hierarchy to deal with the state-dependent methods. Do not implement Strawberry.

3. [20 points] Use the classic Visitor Pattern to write both the Frost class and its superclass. Assume there are only Strawberry and Banana subclasses of Fruit. Do not write the other visitor, Sunshine.

4. [20 points] Draw the UML Sequence diagram for jack Frost visiting chiquita Banana:
    Banana chiquita = new Banana();
    Frost jack     = new Frost();

    <some visitation occurs here> (can’t give it away now!)
    chiquita.getColor();

Start the sequence with the visitation and show the succession of method calls including those into the state pattern that returns the string “black”.

5. A Train has an Engine followed by one or more Cars.

    engine1 <-> car1 <-> car2 <-> car3 <-> car4

The Engine has a method int getMaxPull() which returns the maximum weight the given engine can pull. Each Car has a method int getWeight(). The engineer realizes the train is too heavy to climb a mountain and uses the method void shorten-
Train(int maxWeight) which keeps only the sequence of Cars starting at the Engine whose accumulative weight is less than what the Engine can pull. For instance, if engine1 can only pull the combined weight of car1 and car2, the result of shortenTrain would be:

    engine1 <-> car1 <-> car2    and leaving behind car3 <-> car4

[10 points] Write the UML class diagram for the classes Train, Engine, Car including all above methods.

[15 points] Write the three classes and methods to link a Train together plus the method shortenTrain. Presume the member methods getMaxPull and getWeight are written for you.