1. [20%] Two parts: (a) What should one look for as evidence that the State Pattern would apply in a given situation? and (b) Describe a primary design benefit of the State Pattern.

The SP applies when methods have different ‘behaviors’ (e.g. return values) that are satisfactorily described in terms of the class taking on one of several discrete states. Ideally each state-dependent method have a well-defined value for each state the class might assume. The state pattern does not dictate whether the states or their transitions are externally modifiable or indeed observable. In essence, the SP offers a common interface of methods within the state-dependent class, using dispatching through a hierarchy of classes representing the discrete states.

Design benefits include:

Easy to implement, due to the simplified and unified programming of this pattern. (The state-dependent method simply calls the corresponding method in the given state.) Adherence to a standard programming style, for enhanced understandability. Extensibility, since additional states may be added, just as additional methods that are dependent upon the same states may be added.

2. [30%] The interface TempI is to convert between degrees Fahrenheit (F) and Centigrade (C). Recall $F = (9/5)C + 32$.

```java
public interface TempI {
    public void setFahrenheit(); // converts C to F
    public void setCentigrade(); // converts F to C
    public float convert(float v); // returns converted value
    public boolean freezing(float v); // true if v is freezing
}
```

On the reverse side, write the class Temperature using the pure State Pattern (write the State classes as well), consistent with:
TempI t = new Temperature();  // Temperature implements TempI
float v = 20.0f;  // will be regarded as C then F

t.setCentigrade();
t.freezing(v);  ==> false;  // 20 deg C. is above freezing
t.convert(v);  ==> 68.0;  // and 20 deg C. = 68 deg F.
t.setFahrenheit();  // now convert F. to C.
t.freezing(v);  ==> true;  // 20 deg F. is freezing
t.convert(v);  ==> -6.7;  // and 20 deg F = -6.7 deg C.

Do not use instanceof, or boolean variables, etc. Use conditionals only to test a given float argument for freezing.
public class Temperature implements TempI {
    private Fahrenheit fahrenheit = new Fahrenheit();
    private Centigrade centigrade = new Centigrade();
    private State state = fahrenheit;

    public void setCentigrade() { state = centigrade; }
    public void setFahrenheit() { state = fahrenheit; }
    public float convert(float v) { state.convert(v); }
    public boolean freezing(float v) { state.freezing(v); }
}

abstract public class State {
    abstract float convert(float v);
    abstract boolean freezing(float v);
}

public class Fahrenheit extends State {
    public float convert(float v) { return (v - 32.0f)*5.0f/9.0f; }
    public boolean freezing(float v) { return v <= 32.0f; }
}

public class Centigrade extends State {
    public float convert(float v) { return (9.0f*v/5.0f) + 32.0f; }
    public boolean freezing(float v) { return v <= 0.0f; }
}
3. [50%] Given a simplified ListI:

```java
public interface ListI {
    public void setNext(ListI l);
    public void setItem(Item i);
    public ListI getNext();
    public Item getItem();
    public void acceptVisitor(Visitor v);
}
```

And the familiar implementation of List:

```java
abstract public class List implements ListI { ... }

public class EmptyList extends List {
    public EmptyList() {} // only the default constructor
    public void acceptVisitor(Visitor v) { v.visit(this); }
    ...
}

public class Node extends List {
    public Node(Item i, List l) { ... } // only this constructor
    public void acceptVisitor(Visitor v) { v.visit(this); }
    ...
}
```

Presume abstract Rock implements Item, and that both Diamond and Stone extend Rock.

```java
Item sharon = new Stone();
Item neil = new Diamond();
Item oliver = new Stone();
Item irving = new Stone();
List rocks; // to be { sharon, neil, ... }
```

a) [20%] Write Java code to create the List rocks in the order: sharon, neil, oliver, and irving (that is, sharon should be at the front of the list). Just write a few lines of application code, not the class definitions!!!
b) [30%] **DiamondV** visits with each Node of a List and assumes all Items are instances of Rock (either Stone or Diamond). If it finds a Diamond, it keeps a reference to it and returns. The method get() later returns that reference (or null if no instance of Diamond is found within the List), for example:

**DRIVER CODE:**

```
DiamondV v = new DiamondV();
rocks.acceptVisitor(v);
Diamond d = v.get();
```

Note that acceptVisitor in the List classes is the standard v.vist(this) -- see above. Here is a proposed (maybe buggy?) implementation of DiamondV:

```
public class DiamondV extends Visitor {
    Diamond d;

    public Diamond get() { return d; }

    public visit(Node n) {
        Rock r = (Rock)n.getItem(); // assume cast is successful
        r.acceptVisitor(this);
        if (d == null) {
            List next = n.getNext();
            next.acceptVisitor(this);
        }
    }

    public visit(EmptyList e) {}
    public visit(Stone s) {}
    public visit(Diamond d) { this.d = d; }
}
```

Does it work? Write a sequence diagram for the driver code (above), to show how it would successfully find nil, or use the diagram to reveal a problem in the design, if any. Provide any further discussion as necessary.

**answer:** it works fine. each node gets a visit, the node visits with the visitor then the node passes the visitor to its item (after casting it to a Rock). I could make a joke about casting the first stone, but maybe shouldn’t.