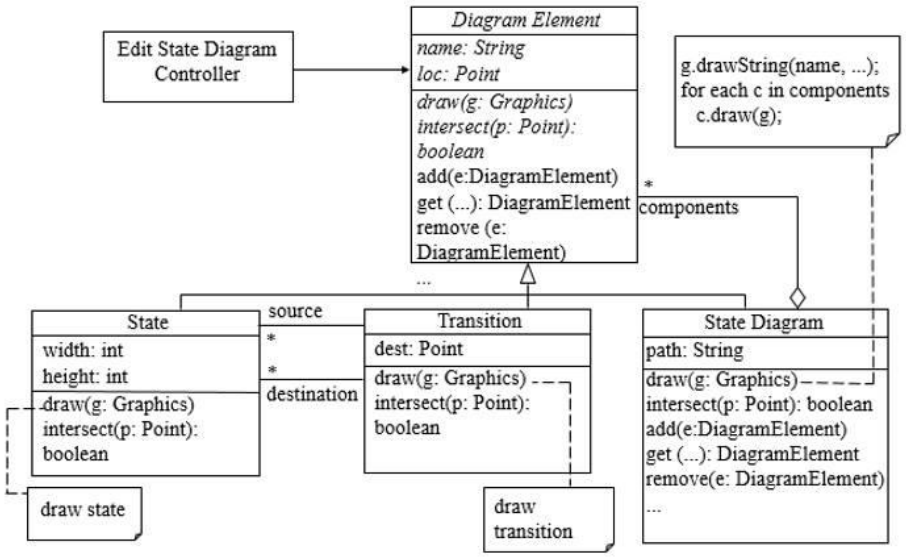
1. The state diagram is an aggregation of State objects and Transition objects. The design of the state diagram editor needs to consider composite states. This means that the design has to consider how to represent complex structures, in which a state object may include other state objects. The following figure shows the representation of a state diagram using the composite pattern. Please write the code and explain the consideration of transparency and safety as detail as possible. **40%**



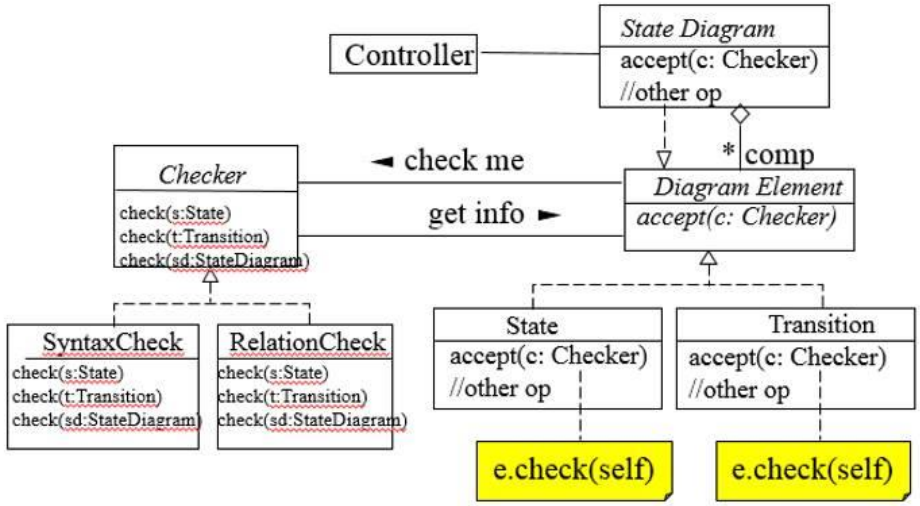

```
import java.awt.Graphics;
import java.util.ArrayList;
import java.awt.Point;

import MementoPattern.Memento;

public abstract class DiagramElement {  //Component and Prototype
        public String name;
        public Point loc;

        public void draw(Graphics g){
                throw new UnsupportedOperationException();
        }

        public boolean intersect(Point p){
                throw new UnsupportedOperationException();
        }

        public void addElement(DiagramElement de){   //stateDiagram
                throw new UnsupportedOperationException();
        }

        public void getElement(DiagramElement de){
                throw new UnsupportedOperationException();
        }

        public void removeElement(DiagramElement de){//stateDiagram
                throw new UnsupportedOperationException();
        }
        public ArrayList<DiagramElement> getChild(){ //stateDiagram
                throw new UnsupportedOperationException();
        }
}
```

```java
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;

public class State extends DiagramElement {

        public int width;
        public int height;

         public State(){
           //Annotation
         }

        @Override
                public void draw(Graphics g){
                //Annotation
        }

        @Override
                public boolean intersect(Point p){
                //Annotation
        }
}
```

```java
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;

public abstract class Transition extends DiagramElement{

        public Point dest;

        public Transition(){
                //Annotation
        }

        @Override
        public void draw(Graphics g){
                //Annotation
        }

        @Override
        public boolean intersect(Point p){
                //Annotation
        }
}
```

```java
import java.awt.Graphics;
import java.util.ArrayList;
import java.util.Iterator;

public class StateDiagram1 extends StateDiagram{

        protected ArrayList<DiagramElement> des = new ArrayList<DiagramElement>();

 public StateDiagram1(){
        super("StateDiagram1");
 }

 @Override
 public void addElement(DiagramElement de){
        System.out.println("----- add element into StateDiagram1 ----- ");
        des.add(de);
 }

 @Override
 public void getElement(DiagramElement de){
        System.out.println("----- get element into StateDiagram1 ----- ");
        get.add(de);
 }

 @Override
 public void removeElement(DiagramElement de){
        System.out.println("StateDiagram1 remove element: " + de.getName());
        des.remove(de);
 }

 @Override
 public ArrayList<DiagramElement> getChild(){
        return this.des;
 }

       @Override
       public void draw(Graphics g){
              System.out.println("----- draw StateDiagram1 ----- ");
              Iterator<DiagramElement> iterator = des.iterator();
              while(iterator.hasNext()){
                     DiagramElement de = iterator.next();
                     de.draw(g);
                     System.out.println(de.getName());
              }
       }

       @Override
       public boolean intersect(Point p){
              //Annotation
       }

}
```

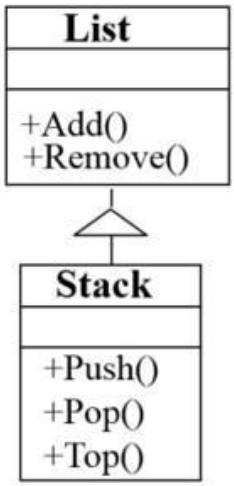
2. The state diagram editor provide analysis capabilities such as checking the syntax and checking the relationship of states and transitions. Without using traditional conditional statements, we will apply the Visitor Pattern as the figure below. Please write the code and explain what double dispatch is used in this pattern. **40%**

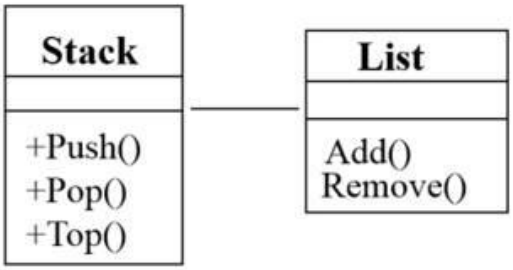



```
import java.util.*;

interface DiagramElement {
    public void accept(Checker c);
}

class StateDiagram implements DiagramElement {
    private ArrayList<DiagramElement> els = new ArrayList<DiagramElement>();

    public void add(DiagramElement e) {
        els.add(e);
    }

    public void accept(Checker c) {
        c.check(this);
        Iterator itr = els.iterator();

        while (itr.hasNext()) {
            ((DiagramElement)itr.next()).accept(c);
        }
    }
}

class State implements DiagramElement {
    public void accept(Checker c) {
        c.check(this);
    }
}

class Transition implements DiagramElement {
    public void accept(Checker c) {
        c.check(this);
    }
}
```

```
interface Checker {
    public void check(State s);
    public void check(Transition t);
    public void check(StateDiagram sd);
}

class SyntaxCheck implements Checker {
    public void check(State s) {
        System.out.println("SyntaxCheck: State " + s);
    }

    public void check(Transition t) {
        System.out.println("SyntaxCheck: Transition " + t);
    }

    public void check(StateDiagram sd) {
        System.out.println("SyntaxCheck: StateDiagram " + sd);
    }
}

class RelationCheck implements Checker {
    public void check(State s) {
        System.out.println("RelationCheck: State " + s);
    }

    public void check(Transition t) {
        System.out.println("RelationCheck: Transition " + t);
    }

    public void check(StateDiagram sd) {
        System.out.println("RelationCheck: StateDiagram " + sd);
    }
}

class Controller {
    public static void main (String[] args) {
        StateDiagram sd = new StateDiagram();

        sd.add(new State());
        sd.add(new State());
        sd.add(new Transition());

        sd.accept(new SyntaxCheck());
        sd.accept(new RelationCheck());
    }
}
```

3. Please compare the following two designs. Which is better? Explain your reasons. **20%**






```
puclic class Stack{
    public List list;
    public Stack(List list){
        this.list = list;
    }
    void push(){
        //put something into stack's top
    }
    void pop(){
        //take out something from stack's top
    }
    void top(){
        //check what is top of stack
    }
    void add(){
        this.list.add();
    }
    void remove(){
        this.list.remove();
    }
}
```

```
public class List{
    public List(){
        //constructor
    }
    void add(){
        //add something into list
    }
    void remove(){
        //take out something from list
    }
}
```

Delegation is better,beacause it can use another class's method when it wants to use, and it is low coupling,high flexible; inheritance is high coupling, and if superclass changes its method, subclass will change method too.