```
public class Task5Driver
{

   public static void main(final String[] arguments)
   {
      Task5Driver driver = new Task5Driver();

      driver.runDynamicTest();
   }


   private void runDynamicTest()
   {
      String id2 = "component2";

      Dimensions dimensions2 = new Dimensions(4, 3, 2);

      Position pivot2 = Position.CENTER;

      DescriptorSpatial descriptor2 = new DescriptorSpatial(dimensions2, pivot2);

      List<LinkageSocket> sockets2 = new ArrayList<>();

      LinkageSocket component2Socket1 = new LinkageSocket("c2.s1", new Position(0, 1.5, 1));

      sockets2.add(component2Socket1);

      ComponentBox component2 = new ComponentBox(id2, descriptor2, sockets2);


      LinkageBall linkage1Ball1 = new LinkageBall("l1.b1", LinkageBall.E_Freedom.FIXED);
      LinkageBall linkage1Ball2 = new LinkageBall("l1.b2", LinkageBall.E_Freedom.FIXED);

      SpanStatic span1 = new SpanStatic("span1", linkage1Ball1, linkage1Ball2);

      span1.bindToSockets(component1Socket1, component2Socket1);


      List<A_Component> components = new ArrayList<>();

      components.add(component1);
      components.add(component2);

      MachineGeneric machine = new MachineGeneric("myMachine", components);
   }
}
```

(your name)

```
public abstract class A_Component implements I_Identifiable, I_XMLable
{
   private final String _id;

   private final DescriptorSpatial _descriptor;

   private final Map<String, LinkageSocket> _sockets = new HashMap<>();

   private Position _position;

   private Attitude _attitudeAbsolute;

   private Attitude _attitudeRelative;

   private final boolean _isRoot;

   private boolean _isCommitted = false;


   public A_Component(final String id, final DescriptorSpatial descriptor, final List<LinkageSocket> sockets)
   {
      this(id, descriptor, sockets, false);
   }


   public A_Component(final String id, final DescriptorSpatial descriptor, final List<LinkageSocket> sockets, final boolean isRoot)
   {
      _id = id;

      _descriptor = descriptor;

      _isRoot = isRoot;

      for (LinkageSocket socket : sockets)
      {

         String idSocket = socket.getID_();

         boolean isDuplicate = _sockets.containsKey(idSocket);

         if (isDuplicate)
         {
            throw new RuntimeException("duplicate socket [" + id + "]");
         }

         socket.setHost(this);

         _sockets.put(idSocket, socket);
      }

      _position = Position.CENTER;

      _attitudeRelative = Attitude.NEUTRAL;
      _attitudeAbsolute = Attitude.NEUTRAL;
   }


   protected void anchor(final Position position, final Attitude attitude)
   {
      setPositionAndAttitude(position, attitude);
```

```
   }

   public void commit()
   {
      _isCommitted = true;

      _attitudeRelative = _attitudeAbsolute;
   }

   public boolean equals(final Object object)
   {
      // ... snipped for Task 5 ...
   }

   public Attitude getAttitude()
   {
      return _attitudeAbsolute;
   }

   public Attitude getAttitudeRelative()
   {
      return _attitudeRelative;
   }

   public String getID_()
   {
      return _id;
   }

   public Position getPosition()
   {
      return _position;
   }

   public Position getPositionPivotAbsolute()
   {
      return _position.add(_descriptor.getPivot());
   }

   public LinkageSocket getSocket(final String id)
   {
      if (!_sockets.containsKey(id))
      {
         throw new RuntimeException("unknown socket [" + id + "]");
      }

      return _sockets.get(id);
   }

   public List<LinkageSocket> getSockets()
   {
      List<LinkageSocket> sockets = new ArrayList<>();

      sockets.addAll(_sockets.values());

      return sockets;
   }

   public DescriptorSpatial getSpatialDescriptor()
   {
      return _descriptor;
   }

   private A_Component getTargetComponent(final A_Span span)
   {
      LinkageBall targetBall = span.getBallTarget();

      LinkageSocket targetSocket = targetBall.getBindingToSocket();

      A_Component targetComponent = targetSocket.getHost();

      return targetComponent;
   }

   public int hashCode()
   {
      return (31 + ((_id == null) ? 0 : _id.hashCode()));
   }

   public boolean hasSocket(final String id)
   {

      return _sockets.containsKey(id);
   }

   public boolean isCommitted()
   {
      return _isCommitted;
   }

   public boolean isRoot()
   {
      return _isRoot;
   }
```

```
   private void resetVisit()
   {
      for (LinkageSocket sourceSocket : _sockets.values())
      {
         if (sourceSocket.isBoundToBall())
         {
            LinkageBall sourceBall = sourceSocket.getBindingToBall();

            A_Span span = sourceBall.getHost();

            if (span.isVisited())
            {
               span.isVisited(false);

               A_Component targetComponent = getTargetComponent(span);

               targetComponent.resetVisit();
            }
         }
      }
   }


   public void setAttitude(final Attitude attitude)
   {
      if (_isCommitted && !_isRoot)
      {
         throw new RuntimeException("operation not valid on committed nonroot component");
      }

      _attitudeAbsolute = attitude;

      update();
   }


   public void setPosition(final Position position)
   {
      if (_isCommitted && !_isRoot)
      {
         throw new RuntimeException("operation not valid on committed nonroot component");
      }

      _position = position;

      update();
   }


   public void setPositionAndAttitude(final Position position, final Attitude attitude)
   {
      _position = position;
      _attitudeAbsolute = attitude;

      update();
   }


   public void update()
   {
      for (LinkageSocket sourceSocket : _sockets.values())
      {
         if (sourceSocket.isBoundToBall())
         {
            LinkageBall sourceBall = sourceSocket.getBindingToBall();

            A_Span span = sourceBall.getHost();

            if (!span.isVisited())
            {
               span.isVisited(true);

               A_Component targetComponent = getTargetComponent(span);

               Position targetComponentPositionPlanned = span.resolveTargetComponent();
               Attitude targetComponentAttitudePlanned = targetComponent.getAttitudeRelative().add(_attitudeAbsolute);

               targetComponent.anchor(targetComponentPositionPlanned, targetComponentAttitudePlanned);
            }
         }
      }

      if (_isRoot)
      {
         resetVisit();
      }
   }
}
```

```
public abstract class A_Linkage<_HOST_ extends I_Identifiable> implements I_Identifiable, I_XMLable
{
   private final String _id;

   private _HOST_ _host;


   public A_Linkage(final String id)
   {
      _id = id;
   }


   public boolean equals(final Object object)
   {
      // ... snipped for Task 5 ...
   }
```

```
   public _HOST_ getHost()
   {
      if (!hasHost())
      {
         throw new RuntimeException("not bound to any host");
      }

      return _host;
   }


   public String getID_()
   {
      return _id;
   }


   public int hashCode()
   {
      return (31 + ((_id == null) ? 0 : _id.hashCode()));
   }


   public boolean hasHost()
   {
      return (_host != null);
   }


   public void setHost(final _HOST_ host)
   {
      if (hasHost())
      {
         throw new RuntimeException("already bound to host [" + host.getID_() + "]");
      }

      _host = host;
   }
```

```
public abstract class A_Machine implements I_Identifiable, I_XMLable, I_Gnuplotable
{
   private final String _id;

   private final Map<String, A_Component> _components = new HashMap<>();

   private final A_Component _componentRoot;


   public A_Machine(final String id, final List<A_Component> components)
   {
      _id = id;

      A_Component componentRoot = null;

      for (A_Component component : components)
      {
         String idComponent = component.getID_();

         boolean isDuplicate = _components.containsKey(idComponent);

         if (isDuplicate)
         {
            throw new RuntimeException("duplicate component [" + id + "]");
         }

         if (component.isRoot())
         {
            if (componentRoot != null)
            {
               throw new RuntimeException("multiple root components in machine [" + id + "]: " + idComponent + " " + componentRoot.getID_());
            }

            componentRoot = component;
         }

         component.commit();

         _components.put(idComponent, component);
      }

      if (componentRoot == null)
      {
         throw new RuntimeException("no root component in machine [" + id + "]");
      }

      _componentRoot = componentRoot;
   }


   public boolean equals(final Object object)
   {
      // ... snipped for Task 5 ...
   }


   public A_Component getComponent(final String id)
   {
      if (!_components.containsKey(id))
      {
         throw new RuntimeException("unknown component [" + id + "]");
      }

      return _components.get(id);
   }


   public A_Component getComponentRoot()
```

```
   {
      return _componentRoot;
   }


   public List<A_Component> getComponents()
   {
      List<A_Component> components = new ArrayList<>();

      components.addAll(_components.values());

      return components;
   }


   public String getID_()
   {
      return _id;
   }


   public int hashCode()
   {
      return (31 + ((_id == null) ? 0 : _id.hashCode()));
   }


   public void update(final Attitude attitude)
   {
      _componentRoot.setAttitude(attitude);
   }


   public void update(final Position position)
   {
      _componentRoot.setPosition(position);
   }


   public void update(final Position position, final Attitude attitude)
   {
      _componentRoot.setPositionAndAttitude(position, attitude);
   }
}
```

```
public abstract class A_Span implements I_Identifiable, I_XMLable, I_Gnuplotable
{
   private final String _id;

   private final LinkageBall _ballSource;

   private final LinkageBall _ballTarget;

   private Position _offsetPivotSourceToBallTarget;

   private boolean _isVisited = false;


   public A_Span(final String id, final LinkageBall ballSource, final LinkageBall ballTarget)
   {
      _id = id;

      _ballSource = ballSource;
      _ballTarget = ballTarget;

      _ballSource.setHost(this);
      _ballTarget.setHost(this);
   }


   private void bind(final LinkageBall ball, final LinkageSocket socket)
   {
      ball.bindToSocket(socket);

      socket.bindToBall(ball);
   }


   public void bindToSockets(final LinkageSocket socketSource, final LinkageSocket socketTarget)
   {
      if (isBoundToSocket())
      {
         throw new RuntimeException("already bound from socket source [" + _ballSource.getBindingToSocket().getID_() + "] to target ["
               + _ballTarget.getBindingToSocket().getID_() + "]");
      }

      bind(_ballSource, socketSource);
      bind(_ballTarget, socketTarget);

      calculateOffset(socketSource, socketTarget);

      commit(socketSource, socketTarget);
   }


   protected void calculateOffset(final LinkageSocket socketSource, final LinkageSocket socketTarget)
   {
      Position positionSocketTarget = socketTarget.getPositionAbsolute();

      A_Component sourceComponent = socketSource.getHost();

      Position sourcePivotAbsolute = sourceComponent.getPositionPivotAbsolute();

      _offsetPivotSourceToBallTarget = positionSocketTarget.subtract(sourcePivotAbsolute);
   }


   private void commit(final LinkageSocket socketSource, final LinkageSocket socketTarget)
```

```
   {
      A_Component componentSource = socketSource.getHost();
      A_Component componentTarget = socketTarget.getHost();

      componentSource.commit();
      componentTarget.commit();
   }


   public boolean equals(final Object object)
   {
      // ... snipped for Task 5 ...
   }


   public LinkageBall getBallSource()
   {
      return _ballSource;
   }


   public LinkageBall getBallTarget()
   {
      return _ballTarget;
   }


   public String getID_()
   {
      return _id;
   }


   public int hashCode()
   {
      return (31 + ((_id == null) ? 0 : _id.hashCode()));
   }


   public boolean isBoundToSocket()
   {
      return _ballSource.isBoundToSocket();
   }


   public boolean isVisited()
   {
      return _isVisited;
   }


   public void isVisited(final boolean isVisited)
   {
      _isVisited = isVisited;
   }


   private Position resolveTargetBall()
   {
      A_Component sourceComponent = getBallSource().getBindingToSocket().getHost();

      Attitude sourceAttitude = sourceComponent.getAttitude();

      Position sourcePositionPivotAbsolute = sourceComponent.getPositionPivotAbsolute();

      Position targetPositionTranslated = sourcePositionPivotAbsolute.add(_offsetPivotSourceToBallTarget);

      Position targetPositionRotated = targetPositionTranslated.rotate(sourcePositionPivotAbsolute, sourceAttitude);

      return targetPositionRotated;
   }


   public abstract Position resolveTargetComponent();
}
```

```
public class Attitude implements I_XMLable
{
   public static final Attitude NEUTRAL = new Attitude(Yaw.NEUTRAL, Pitch.NEUTRAL, Roll.NEUTRAL);

   private final Yaw _yaw;

   private final Pitch _pitch;

   private final Roll _roll;


   public Attitude()
   {
      _yaw = new Yaw();
      _pitch = new Pitch();
      _roll = new Roll();
   }


   public Attitude(final Pitch pitch)
   {
      this(new Yaw(), pitch, new Roll());
   }


   public Attitude(final Pitch pitch, final Roll roll)
   {
      this(new Yaw(), pitch, roll);
   }
```

```
public Attitude(final Roll roll)
{
   this(new Yaw(), new Pitch(), roll);
}


public Attitude(final Yaw yaw)
{
   this(yaw, new Pitch(), new Roll());
}


public Attitude(final Yaw yaw, final Pitch pitch)
{
   this(yaw, pitch, new Roll());
}


public Attitude(final Yaw yaw, final Pitch pitch, final Roll roll)
{
   _yaw = yaw;
   _pitch = pitch;
   _roll = roll;
}


public Attitude(final Yaw yaw, final Roll roll)
{
   this(yaw, new Pitch(), roll);
}


public Attitude add(final Attitude attitude)
{
   Yaw yaw = _yaw.add(attitude._yaw);

   Pitch pitch = _pitch.add(attitude._pitch);

   Roll roll = _roll.add(attitude._roll);

   return new Attitude(yaw, pitch, roll);
}


public Pitch getPitch()
{
   return _pitch;
}


public Roll getRoll()
{
   return _roll;
}


public Yaw getYaw()
{
   return _yaw;
}


public Attitude setAttitude(final Pitch pitch, final Roll roll)
{
   return new Attitude(_yaw, pitch, roll);
}


public Attitude setAttitude(final Yaw yaw, final Pitch pitch)
{
   return new Attitude(yaw, pitch, _roll);
}


public Attitude setAttitude(final Yaw yaw, final Roll roll)
{
   return new Attitude(yaw, _pitch, roll);
}


public Attitude setPitch(final Pitch pitch)
{
   return new Attitude(_yaw, pitch, _roll);
}


public Attitude setRoll(final Roll roll)
{
   return new Attitude(_yaw, _pitch, roll);
}


public Attitude setYaw(final Yaw yaw)
{
   return new Attitude(yaw, _pitch, _roll);
}


public Attitude subtract(final Attitude attitude)
{
   Yaw yaw = _yaw.subtract(attitude._yaw);

   Pitch pitch = _pitch.subtract(attitude._pitch);

   Roll roll = _roll.subtract(attitude._roll);

   return new Attitude(yaw, pitch, roll);
}
```

```
public class Bearing implements I_XMLable
{
   private final Position _origin;

   private final Vector _vector;


   public Bearing(final Position origin, final Vector vector)
   {
      _origin = origin;
      _vector = vector;
   }


   public Bearing add(final Bearing bearing)
   {
      Position origin = _origin.add(bearing.getOrigin());

      Vector vector = _vector.add(bearing.getVector());

      return new Bearing(origin, vector);
   }


   public Position getOrigin()
   {
      return _origin;
   }


   public Vector getVector()
   {
      return _vector;
   }


   public Position resolveTarget()
   {
      return _vector.resolveTarget(_origin);
   }


   public Bearing subtract(final Bearing bearing)
   {
      Position origin = _origin.subtract(bearing.getOrigin());

      Vector vector = _vector.subtract(bearing.getVector());

      return new Bearing(origin, vector);
   }


   public Bearing updateVector(final Attitude attitude)
   {
      Vector vector = _vector.updateAttitude(attitude);

      return new Bearing(_origin, vector);
   }


   public Bearing updateVector(final Vector vector)
   {
      return new Bearing(_origin, vector);
   }
}
```

```
public class ComponentBox extends A_Component
{
   private final DescriptorBox _boxBase;

   private DescriptorBox _boxCurrent;


   public ComponentBox(final String id, final DescriptorSpatial descriptor, final List<LinkageSocket> sockets)
   {
      this(id, descriptor, sockets, false);
   }


   public ComponentBox(final String id, final DescriptorSpatial descriptor, final List<LinkageSocket> sockets, final boolean isRoot)
   {
      super(id, descriptor, sockets, isRoot);

      Position pivot = descriptor.getPivot();

      _boxBase = generateBox(Position.CENTER, pivot);

      _boxCurrent = _boxBase;
   }


   private DescriptorBox generateBox(final Position base, final Position pivot)
   {
      DescriptorSpatial descriptor = getSpatialDescriptor();

      Dimensions dimensions = descriptor.getDimensions();

      Position origin = dimensions.getOrigin();

      double xM = -origin.getX();
      double xP = +origin.getX();

      double yM = -origin.getY();
      double yP = +origin.getY();

      double zM = -origin.getZ();
```

```
      double zP = +origin.getZ();

      Position cornerTopFrontLeft = new Position("top front left", xM, yM, zP);
      Position cornerTopFrontRight = new Position("top front right", xP, yM, zP);
      Position cornerTopBackRight = new Position("top back right", xP, yP, zP);
      Position cornerTopBackLeft = new Position("top back left", xM, yP, zP);

      Position cornerBottomFrontLeft = new Position("bottom from left", xM, yM, zM);
      Position cornerBottomFrontRight = new Position("bottom front right", xP, yM, zM);
      Position cornerBottomBackRight = new Position("bottom back right", xP, yP, zM);
      Position cornerBottomBackLeft = new Position("bottom back left", xM, yP, zM);

      List<LinkageSocket> sockets = getSockets();

      DescriptorBox box = new DescriptorBox(pivot,
                                            cornerTopFrontLeft,
                                            cornerTopFrontRight,
                                            cornerTopBackRight,
                                            cornerTopBackLeft,
                                            cornerBottomFrontLeft,
                                            cornerBottomFrontRight,
                                            cornerBottomBackRight,
                                            cornerBottomBackLeft,
                                            sockets);

      DescriptorBox boxAnchored = box.translate(base.negate());

      return boxAnchored;
   }


   public DescriptorBox getBoxBase()
   {
      return _boxBase;
   }


   public DescriptorBox getBoxCurrent()
   {
      return _boxCurrent;
   }


   public void update()
   {
      super.update();

      Attitude attitude = getAttitude();

      _boxCurrent = _boxBase.rotate(attitude);

      Position position = getPosition();

      _boxCurrent = _boxCurrent.translate(position);
   }
}
```

```
public class DescriptorBox implements I_XMLable, I_Gnuplotable
{
   private final Position _pivot;

   private final Position _cornerTopFrontLeft;

   private final Position _cornerTopFrontRight;

   private final Position _cornerTopBackRight;

   private final Position _cornerTopBackLeft;

   private final Position _cornerBottomFrontLeft;

   private final Position _cornerBottomFrontRight;

   private final Position _cornerBottomBackRight;

   private final Position _cornerBottomBackLeft;

   private final List<LinkageSocket> _sockets;


   public DescriptorBox(final Position pivot,
                        final Position cornerTopFrontLeft,
                        final Position cornerTopFrontRight,
                        final Position cornerTopBackRight,
                        final Position cornerTopBackLeft,
                        final Position cornerBottomFrontLeft,
                        final Position cornerBottomFrontRight,
                        final Position cornerBottomBackRight,
                        final Position cornerBottomBackLeft,
                        final List<LinkageSocket> sockets)
   {
                     cornerTopFrontLeft,
                     cornerTopFrontRight,
                     cornerTopBackRight,
                     cornerTopBackLeft,
                     cornerBottomFrontLeft,
                     cornerBottomFrontRight,
                     cornerBottomBackRight,
                     cornerBottomBackLeft,
                     sockets);

      _pivot = pivot;

      _cornerTopFrontLeft = cornerTopFrontLeft;
      _cornerTopFrontRight = cornerTopFrontRight;
      _cornerTopBackRight = cornerTopBackRight;
      _cornerTopBackLeft = cornerTopBackLeft;
```

```
      _cornerBottomFrontLeft = cornerBottomFrontLeft;
      _cornerBottomFrontRight = cornerBottomFrontRight;
      _cornerBottomBackRight = cornerBottomBackRight;
      _cornerBottomBackLeft = cornerBottomBackLeft;

      _sockets = Collections.unmodifiableList(sockets);
   }


   private void buildString(final StringBuilder stream, final Position position, final String description, final int newlineCount)
   {
      stream.append(position.toGnuplot_() + "   # " + description);

      for (int iNewline = 0; iNewline < newlineCount; ++iNewline)
      {
         stream.append(NEWLINE);
      }
   }


   public Position getCornerBottomBackLeft()
   {
      return _cornerBottomBackLeft;
   }


   public Position getCornerBottomBackRight()
   {
      return _cornerBottomBackRight;
   }


   public Position getCornerBottomFrontLeft()
   {
      return _cornerBottomFrontLeft;
   }


   public Position getCornerBottomFrontRight()
   {
      return _cornerBottomFrontRight;
   }


   public Position[] getCorners()
   {
      return new Position[]
      { _cornerTopFrontLeft, _cornerTopFrontRight, _cornerTopBackRight, _cornerTopBackLeft, _cornerBottomFrontLeft, _cornerBottomFrontRight,
            _cornerBottomBackRight, _cornerBottomBackLeft };
   }


   public Position getCornerTopBackLeft()
   {
      return _cornerTopBackLeft;
   }


   public Position getCornerTopBackRight()
   {
      return _cornerTopBackRight;
   }


   public Position getCornerTopFrontLeft()
   {
      return _cornerTopFrontLeft;
   }


   public Position getCornerTopFrontRight()
   {
      return _cornerTopFrontRight;
   }


   public Position getPivot()
   {
      return _pivot;
   }


   public List<LinkageSocket> getSockets()
   {
      return _sockets;
   }


   public DescriptorBox rotate(final Attitude attitude)
   {
      return rotate(attitude, _pivot);
   }


   public DescriptorBox rotate(final Attitude attitude, final Position pivot)
   {
      Position pivot2 = pivot.rotate(pivot, attitude);

      Position cornerTopFrontLeft = _cornerTopFrontLeft.rotate(pivot, attitude);
      Position cornerTopFrontRight = _cornerTopFrontRight.rotate(pivot, attitude);
      Position cornerTopBackRight = _cornerTopBackRight.rotate(pivot, attitude);
      Position cornerTopBackLeft = _cornerTopBackLeft.rotate(pivot, attitude);
      Position cornerBottomFrontLeft = _cornerBottomFrontLeft.rotate(pivot, attitude);
      Position cornerBottomFrontRight = _cornerBottomFrontRight.rotate(pivot, attitude);
      Position cornerBottomBackRight = _cornerBottomBackRight.rotate(pivot, attitude);
      Position cornerBottomBackLeft = _cornerBottomBackLeft.rotate(pivot, attitude);

      for (LinkageSocket socket : _sockets)
      {
         Position position = socket.getPositionRelative();
```

```
         Position positionRotated = position.rotate(pivot, attitude);

         socket.setPositionAbsolute(positionRotated);
      }

      return new DescriptorBox(pivot2,
                               cornerTopFrontLeft,
                               cornerTopFrontRight,
                               cornerTopBackRight,
                               cornerTopBackLeft,
                               cornerBottomFrontLeft,
                               cornerBottomFrontRight,
                               cornerBottomBackRight,
                               cornerBottomBackLeft,
                               _sockets);
   }


   public DescriptorBox translate(final Position offset)
   {
      Position pivot = _pivot.add(offset);

      Position cornerTopFrontLeft = _cornerTopFrontLeft.add(offset);
      Position cornerTopFrontRight = _cornerTopFrontRight.add(offset);
      Position cornerTopBackRight = _cornerTopBackRight.add(offset);
      Position cornerTopBackLeft = _cornerTopBackLeft.add(offset);

      Position cornerBottomFrontLeft = _cornerBottomFrontLeft.add(offset);
      Position cornerBottomFrontRight = _cornerBottomFrontRight.add(offset);
      Position cornerBottomBackRight = _cornerBottomBackRight.add(offset);
      Position cornerBottomBackLeft = _cornerBottomBackLeft.add(offset);

      for (LinkageSocket socket : _sockets)
      {
         Position position = (socket.hasPositionAbsolute() ? socket.getPositionAbsolute() : socket.getPositionRelative());

         Position positionTranslated = position.add(offset);

         socket.setPositionAbsolute(positionTranslated);
      }

      return new DescriptorBox(pivot,
                               cornerTopFrontLeft,
                               cornerTopFrontRight,
                               cornerTopBackRight,
                               cornerTopBackLeft,
                               cornerBottomFrontLeft,
                               cornerBottomFrontRight,
                               cornerBottomBackRight,
                               cornerBottomBackLeft,
                               _sockets);
   }
}
```

```
public class DescriptorSpatial implements I_XMLable
{
   private final Position _pivot;

   private final Dimensions _dimensions;


   public DescriptorSpatial(final Dimensions dimensions, final Position pivot)
   {
      _dimensions = dimensions;
      _pivot = pivot;
   }


   public Dimensions getDimensions()
   {
      return _dimensions;
   }


   public Position getPivot()
   {
      return _pivot;
   }
```

```
public class Dimensions implements I_XMLable
{
   public static final Position ORIGIN = new Position(0, 0, 0);

   private final double _width;

   private final double _depth;

   private final double _height;


   public Dimensions(final double width, final double depth, final double height)
   {
      if (width <= 0)
      {
         throw new IllegalArgumentException("illegal width: " + width);
      }

      if (depth <= 0)
      {
         throw new IllegalArgumentException("illegal depth: " + depth);
      }

      if (height <= 0)
      {
         throw new IllegalArgumentException("illegal height: " + height);
```

```
      }

      _width = width;
      _depth = depth;
      _height = height;
   }


   public Dimensions add(final Dimensions dimensions)
   {
      double height = (_height + dimensions._height);
      double width = (_width + dimensions._width);
      double depth = (_depth + dimensions._depth);

      return new Dimensions(width, depth, height);
   }


   public double getDepth()
   {
      return _depth;
   }


   public double getHeight()
   {
      return _height;
   }


   public Position getOrigin()
   {
      double x = (_width / 2);
      double y = (_depth / 2);
      double z = (_height / 2);

      return new Position(x, y, z);
   }


   public double getWidth()
   {
      return _width;
   }


   public boolean isWithinFrame(final Position position)
   {
      double shiftedWidth = (_width / 2);
      double shiftedDepth = (_depth / 2);
      double shiftedHeight = (_height / 2);

      double x = position.getX();
      double y = position.getY();
      double z = position.getZ();

      return ((x < -shiftedWidth) || (x > +shiftedWidth) || (y < -shiftedDepth) || (y > +shiftedDepth) || (z < -shiftedHeight) || (z > +shiftedHeight));
   }


   public Dimensions subtract(final Dimensions dimensions)
   {
      double width = (_width - dimensions._width);
      double depth = (_depth - dimensions._depth);
      double height = (_height - dimensions._height);

      return new Dimensions(width, depth, height);
   }
```

```
public class Distance implements I_XMLable, Comparable<Distance>
{
   private final double _distance;

   public Distance(final double distance)
   {
      if (distance < 0)
      {
         throw new IllegalArgumentException("invalid distance: " + distance);

      }
      _distance = distance;
   }


   public Distance add(final Distance distance)
   {

      return new Distance(_distance + distance.getValue());
   }


   public int compareTo(final Distance distance)
   {
      return Double.compare(_distance, distance._distance);
   }


   public double getValue()
   {
      return _distance;
   }


   public Distance subtract(final Distance distance)
   {
      return new Distance(_distance - distance.getValue());
   }
```

```
public class LinkageBall extends A_Linkage<A_Span>
{
   public enum E_Freedom
   {
      FIXED,

      FREE_X,

      FREE_Y,

      FREE_Z
   }

   private final E_Freedom _freedom;

   private LinkageSocket _socket;


   public LinkageBall(final String id, final E_Freedom freedom)
   {
      super(id);

      _freedom = freedom;
   }


   protected void bindToSocket(final LinkageSocket socket)
   {
      if (isBoundToSocket())
      {
         throw new RuntimeException("already bound to socket [" + _socket.getID_() + "]");
      }

      _socket = socket;
   }


   public LinkageSocket getBindingToSocket()
   {
      if (!isBoundToSocket())
      {
         throw new RuntimeException("no bound to any socket");

      }

      return _socket;
   }


   public E_Freedom getFreedom()
   {
      return _freedom;
   }


   public boolean isBoundToSocket()
   {
      return (_socket != null);
   }
```

```
public class LinkageSocket extends A_Linkage<A_Component>
{
   private final Position _positionRelative;

   private Position _positionAbsolute;

   private LinkageBall _ball;


   public LinkageSocket(final String id, final Position positionRelative)
   {
      super(id);

      _positionRelative = positionRelative;
   }


   protected void bindToBall(final LinkageBall ball)
   {
      if (isBoundToBall())
      {
         throw new RuntimeException("already bound to ball [" + _ball.getID_() + "]");
      }

      _ball = ball;
   }


   public LinkageBall getBindingToBall()
   {
      if (!isBoundToBall())
      {
         throw new RuntimeException("not bound to any ball");

      }

      return _ball;
   }


   public Position getPositionAbsolute()
   {
      if (!hasPositionAbsolute())
      {
         throw new RuntimeException("no absolute position set on socket [" + getID_() + "]");
      }
```

```
      return _positionAbsolute;
   }


   public Position getPositionRelative()
   {
      return _positionRelative;
   }


   public boolean hasPositionAbsolute()
   {
      return (_positionAbsolute != null);
   }


   public boolean isBoundToBall()
   {
      return (_ball != null);
   }


   public void setHost(final A_Component host)
   {
      super.setHost(host);
   }


   public void setPositionAbsolute(final Position position)
   {
      _positionAbsolute = position;
   }
```

```
public class MachineGeneric extends A_Machine
{

   public MachineGeneric(final String id, final List<A_Component> components)
   {
      super(id, components);
   }
```

```
public class Pitch implements I_XMLable
{
   public static final Pitch NEUTRAL = new Pitch(0);

   private static final double ANGLE_MIN = -180;

   private static final double ANGLE_MAX = +180;


   public static boolean isValid(final double angle)
   {
      return ((angle >= ANGLE_MIN) && (angle <= ANGLE_MAX));
   }


   public static double normalize(final double angle)
   {
      double angle2 = angle;

      if (angle < ANGLE_MIN)
      {
         angle2 = (ANGLE_MAX - (-angle % ANGLE_MAX));
      }
      else if (angle >= ANGLE_MAX)
      {
         angle2 %= ANGLE_MAX;
      }

      return angle2;
   }

   private final double _angle;


   public Pitch()
   {
      _angle = 0;
   }


   public Pitch(final double angle)
   {
      validate(angle);

      _angle = angle;
   }


   public Pitch add(final Pitch pitch)
   {
      return new Pitch(_angle + pitch._angle);
   }


   public double getAngleDegrees()
   {
      return _angle;
   }


   public double getAngleRadians()
   {
      return Math.toRadians(_angle);
   }
```

```
   public Pitch negate()
   {
      double angle = normalize(_angle + 180);

      return new Pitch(angle);
   }


   public Pitch subtract(final Pitch pitch)
   {
      return new Pitch(_angle - pitch._angle);
   }


   private void validate(final double angle)
   {
      if (!isValid(angle))
      {
         throw new IllegalArgumentException("invalid pitch: " + angle);
      }
   }
}
```

```
public class Position implements I_XMLable, I_Gnuplotable
{
   public static final Position CENTER = new Position(0, 0, 0);

   private String _name;

   private final double _x;

   private final double _y;

   private final double _z;


   public Position(final double x, final double y, final double z)
   {
      _name = null;

      _x = x;
      _y = y;
      _z = z;
   }


   public Position(final String name, final double x, final double y, final double z)
   {
      _name = name;

      _x = x;
      _y = y;
      _z = z;
   }


   public Position add(final Position position)
   {
      double x = (_x + position._x);
      double y = (_y + position._y);
      double z = (_z + position._z);

      return new Position(_name, x, y, z);
   }


   public Attitude calculateAttitude(final Position position)
   {
      Yaw yaw = calculateYaw(position);

      Pitch pitch = calculatePitch(position);

      return new Attitude(yaw, pitch);
   }


   public Bearing calculateBearing(final Position position)
   {
      Vector vector = calculateVector(position);

      return new Bearing(this, vector);
   }


   public Distance calculateDistance(final Position position)
   {
      Position delta = subtract(position);

      double deltaX = delta.getX();
      double deltaY = delta.getY();
      double deltaZ = delta.getZ();

      return new Distance(distance);
   }


   public Pitch calculatePitch(final Position position)
   {
      double deltaX = (_x - position._x);
      double deltaY = (_y - position._y);
      double deltaZ = (_z - position._z);

      double angle = 0;

      if (distance != 0)
      {
```

```
      angle = (90 - Math.toDegrees(Math.acos(-deltaZ / distance)));
   }

   return new Pitch(angle);
}


public Position calculatePosition(final Vector vector)
{
   return vector.resolveTarget(this);
}


public Vector calculateVector(final Position position)
{
   return new Vector(this, position);
}


public Yaw calculateYaw(final Position position)
{
   double deltaX = (_x - position._x);
   double deltaY = (_y - position._y);

   double angle = (Math.toDegrees(Math.atan2(-deltaY, deltaX)) - 90);

   angle = Yaw.normalize(angle);

   return new Yaw(angle);
}


public boolean equals(final Object object)
{
   // ... snipped for Task 5 ...
}


public String getName()
{
   if (!hasName())
   {
      throw new RuntimeException("no name set");
   }

   return _name;
}


public double getX()
{
   return _x;
}


public double getY()
{
   return _y;
}


public double getZ()
{
   return _z;
}


public int hashCode()
{
}


public boolean hasName()
{
   return (_name != null);
}


public Position negate()
{
   return new Position(_name, (-_x), (-_y), (-_z));
}


public Position rotate(final Position pivot, final Attitude attitude)
{
   Position pointRoll = rotate(pivot, attitude.getRoll());

   Position pointPitch = pointRoll.rotate(pivot, attitude.getPitch());

   Position pointYaw = pointPitch.rotate(pivot, attitude.getYaw());

   return pointYaw;
}


public Position rotate(final Position pivot, final Pitch pitch)
{
   Position pointTranslatedToOrigin = translate(pivot.negate());

   double pitch2 = -pitch.getAngleRadians();

   double sinPitch = Math.sin(pitch2);
   double cosPitch = Math.cos(pitch2);

   Position pointRotated = new Position(_name, pointTranslatedToOrigin._x, y, z);

   Position pointTranslatedFromOrigin = pointRotated.translate(pivot);

   return pointTranslatedFromOrigin;
```

```
   }


   public Position rotate(final Position pivot, final Roll roll)
   {
      Position pointTranslatedToOrigin = translate(pivot.negate());

      double roll2 = roll.getAngleRadians();

      double sinRoll = Math.sin(roll2);
      double cosRoll = Math.cos(roll2);

      Position pointRotated = new Position(_name, x, pointTranslatedToOrigin._y, z);

      Position pointTranslatedFromOrigin = pointRotated.translate(pivot);

      return pointTranslatedFromOrigin;
   }


   public Position rotate(final Position pivot, final Yaw yaw)
   {
      Position pointTranslatedToOrigin = translate(pivot.negate());

      double yaw2 = -yaw.getAngleRadians();

      double sinYaw = Math.sin(yaw2);
      double cosYaw = Math.cos(yaw2);

      Position pointRotated = new Position(_name, x, y, pointTranslatedToOrigin._z);

      Position pointTranslatedFromOrigin = pointRotated.translate(pivot);

      return pointTranslatedFromOrigin;
   }


   public void setName(final String name)
   {
      _name = name;
   }


   public Position subtract(final Position position)
   {
      double x = (_x - position._x);
      double y = (_y - position._y);
      double z = (_z - position._z);

      return new Position(_name, x, y, z);
   }


   public Position translate(final Position offset)
   {
      return add(offset);
   }
}
```

```
public class Roll implements I_XMLable
{
   public static final Roll NEUTRAL = new Roll(0);

   private static final double ANGLE_MIN = -180;

   private static final double ANGLE_MAX = +180;


   public static boolean isValid(final double angle)
   {
      return ((angle >= ANGLE_MIN) && (angle <= ANGLE_MAX));
   }


   public static double normalize(final double angle)
   {
      double angle2 = angle;

      if (angle < ANGLE_MIN)
      {
         angle2 = (ANGLE_MAX - (-angle % ANGLE_MAX));
      }
      else if (angle >= ANGLE_MAX)
      {
         angle2 %= ANGLE_MAX;
      }

      return angle2;
   }

   private final double _angle;


   public Roll()
   {
      _angle = 0;
   }


   public Roll(final double angle)
   {
      validate(angle);

      _angle = angle;
   }


   public Roll add(final Roll roll)
```

```
   {
      return new Roll(_angle + roll._angle);
   }


   public double getAngleDegrees()
   {
      return _angle;
   }


   public double getAngleRadians()
   {
      return Math.toRadians(_angle);
   }


   public Roll negate()
   {
      double angle = normalize(_angle + 180);

      return new Roll(angle);
   }


   public Roll subtract(final Roll roll)
   {
      return new Roll(_angle - roll._angle);
   }


   private void validate(final double angle)
   {
      if (!isValid(angle))
      {
         throw new IllegalArgumentException("invalid roll: " + angle);
      }
   }
}
```

```
public class SpanDynamic extends A_Span
{
   public SpanDynamic(final String id, final LinkageBall ballSource, final LinkageBall ballTarget)
   {
      super(id, ballSource, ballTarget);

      if (ballSource.getFreedom() != ballTarget.getFreedom())
      {
         throw new RuntimeException("both balls must have same freedom in current version: " + ballSource.getFreedom() + " != " + ballTarget.getFreedom());
      }
   }

   // [xxx motion stuff]

   public Position resolveTargetComponent()
   {
      return null;
   }
```

```
public class SpanStatic extends A_Span
{
   private Position _offsetPivotSourceToOriginTarget;

   public SpanStatic(final String id, final LinkageBall ballSource, final LinkageBall ballTarget)
   {
      super(id, ballSource, ballTarget);

      if ((ballSource.getFreedom() != LinkageBall.E_Freedom.FIXED) || (ballTarget.getFreedom() != LinkageBall.E_Freedom.FIXED))
      {
         throw new RuntimeException("both balls must be fixed in current version: " + ballSource.getFreedom() + " / " + ballTarget.getFreedom());
      }
   }


   protected void calculateOffset(final LinkageSocket socketSource, final LinkageSocket socketTarget)
   {
      super.calculateOffset(socketSource, socketTarget);

      A_Component sourceComponent = socketSource.getHost();

      Position positionTarget = socketTarget.getHost().getPosition();

      Position sourcePivotAbsolute = sourceComponent.getPositionPivotAbsolute();

      _offsetPivotSourceToOriginTarget = positionTarget.subtract(sourcePivotAbsolute);
   }


   public Position resolveTargetComponent()
   {
      A_Component sourceComponent = getBallSource().getBindingToSocket().getHost();

      Attitude sourceAttitude = sourceComponent.getAttitude();

      Position sourcePositionPivotAbsolute = sourceComponent.getPositionPivotAbsolute();

      Position targetPositionTranslated = sourcePositionPivotAbsolute.add(_offsetPivotSourceToOriginTarget);

      Position targetPositionRotated = targetPositionTranslated.rotate(sourcePositionPivotAbsolute, sourceAttitude);

      return targetPositionRotated;
   }
```

```
public class Vector implements I_XMLable
{
   private final Attitude _attitude;

   private final Distance _distance;


   public Vector(final Attitude attitude, final Distance distance)
   {
      _attitude = attitude;
      _distance = distance;
   }


   public Vector(final Position position1, final Position position2)
   {
      _attitude = position1.calculateAttitude(position2);
      _distance = position1.calculateDistance(position2);
   }


   public Vector add(final Attitude attitude)
   {
      Attitude attitudeNew = _attitude.add(attitude);

      return new Vector(attitudeNew, _distance);
   }


   public Vector add(final Vector vector)
   {
      Attitude attitude = _attitude.add(vector.getAttitude());

      Distance distance = _distance.add(vector.getDistance());

      return new Vector(attitude, distance);
   }


   public Attitude getAttitude()
   {
      return _attitude;
   }


   public Distance getDistance()
   {
      return _distance;
   }


   public Vector negate()
   {
      Yaw yawNew = _attitude.getYaw().negate();

      Pitch pitchNew = _attitude.getPitch().negate();

      Attitude attitudeNew = new Attitude(yawNew, pitchNew);

      return new Vector(attitudeNew, _distance);
   }


   public Position resolveTarget(final Position source)
   {
      double sourceX = source.getX();
      double sourceY = source.getY();
      double sourceZ = source.getZ();

      double yaw = Math.toRadians(-_attitude.getYaw().getAngleDegrees() + 90);
      double pitch = Math.toRadians(90 - _attitude.getPitch().getAngleDegrees());

      double distance = _distance.getValue();

      return new Position(targetX, targetY, targetZ);
   }


   public Vector subtract(final Attitude attitude)
   {
      Attitude attitudeNew = _attitude.subtract(attitude);

      return new Vector(attitudeNew, _distance);
   }


   public Vector subtract(final Vector vector)
   {
      Attitude attitude = _attitude.subtract(vector.getAttitude());

      Distance distance = _distance.subtract(vector.getDistance());

      return new Vector(attitude, distance);
   }


   public Vector updateAttitude(final Attitude attitude)
   {
      Attitude attitudeNew = _attitude.add(attitude);

      return new Vector(attitudeNew, _distance);
   }
}
```

```
public class Yaw implements I_XMLable
{
   public static final Yaw NEUTRAL = new Yaw(0);

   private static final double ANGLE_MIN = 0;

   private static final double ANGLE_MAX = 360;


   public static boolean isValid(final double angle)
   {
      return ((angle >= ANGLE_MIN) && (angle < ANGLE_MAX));
   }


   public static double normalize(final double angle)
   {
      double angle2 = angle;

      if (angle < ANGLE_MIN)
      {
         angle2 = (ANGLE_MAX - (-angle % ANGLE_MAX));
      }
      else if (angle >= ANGLE_MAX)
      {
         angle2 %= ANGLE_MAX;
      }

      return angle2;
   }

   private final double _angle;


   public Yaw()
   {
      _angle = 0;
   }


   public Yaw(final double angle)
   {
      _angle = normalize(angle);
   }


   public Yaw add(final Yaw yaw)
   {
      double angle = normalize(_angle + yaw._angle);

      return new Yaw(angle);
   }


   public double getAngleDegrees()
   {
      return _angle;
   }


   public double getAngleRadians()
   {
      return Math.toRadians(_angle);
   }


   public Yaw negate()
   {
      double angle = normalize(_angle + 180);

      return new Yaw(angle);
   }


   public Yaw subtract(final Yaw yaw)
   {
      double angle = normalize(_angle - yaw._angle);

      return new Yaw(angle);
   }
}
```