```java
package fi.paatti.research.paattiapplication.views.eventview.tools;


import com.vaadin.data.Property;
import com.vaadin.data.util.BeanItem;
import java.awt.Color;
import java.awt.Graphics;
import java.io.Serializable;
import java.util.LinkedList;


/**
 * NodeFactory handles the creation of the TreeNodes used in the event tree.
 *
 * @author Tapio Keränen, t.tapio.keranen@student.jyu.fi
 */
public class NodeFactory implements Serializable {

  // Default color for the connector between two nodes.
  private static final Color CONNECTOR_COLOR = Color.red;
  // Default node size.
  private static final int NODE_WIDTH = 50, NODE_HEIGHT = 50;
  // Running id for new nodes.
  private int id;

  /**
   * The enum that contains the node types used by the node factory.
   *
   * @author Tapio Keränen, t.tapio.keranen@student.jyu.fi
   */
  public static enum NODE_TYPE {
    ROOT, DEFAULT, RADIO
  }

  /**
   * NodeFactory constructor.
   */
  public NodeFactory() {}

  /**
   * Creates a new node of the given type and places it to the given point
   * (x,y).
   *
   * @param type the enum type of the new node.
   * @param x the x coordinate.
   * @param y the y coordinate.
```

```
   * @return the new node of the specified type.
   */
  public TreeNode createNode(NODE_TYPE type, int x, int y) {
    switch (type) {
    case ROOT:
      return new RootNode(id++, x, y);

    case RADIO:
      return new RadioNode(id++, x, y);

    default:
      return new TextNode(id++, x, y);
    }
  }

  /**
   * Sets the factory's running id back to zero.
   */
  public void reset() {
    id = 0;
  }

  /**
   * TreeNode is the base class extended by all the nodes in an event tree.
   * The base class contains the basic functionality for things like setting
   * and getting the node's description, content, etc.
   */
  public class TreeNode extends Clickable {

    // Identifier in the current event tree.
    private int id;
    // Identifier in the database
    private int realId;
    // Description (or "assignment") for the node.
    private String description;
    // Node contents
    private String content;
    // Is the node active or not (should the border color be red or black).
    private boolean active;
    // Vaadin bean that gives access to the node's properties.
    private BeanItem bean;
    // The type of the node based on values listed in db table TASKTYPE.
    private int type;

    // A list of all the connected parents.
    protected LinkedList<TreeNode> parents;
```

```
    // A list of all the connected children.
    protected LinkedList<TreeNode> children;
    // A color variable to help identify nodes of different type.
    protected Color baseColor;

    /**
     * TreeNode constructor.
     *
     * @param id node id
     * @param x the x coordinate
     * @param y the y coordinate
     */
    private TreeNode(int id, int x, int y) {
      this.id = id;
      this.x = x;
      this.y = y;
      this.width = NODE_WIDTH;
      this.height = NODE_HEIGHT;

      realId = -1;

      parents = new LinkedList<TreeNode>();
      children = new LinkedList<TreeNode>();

      bean = new BeanItem(this);
    }

    /**
     * Returns the node's tree id.
     *
     * @return id
     */
    public int getId() {
      return id;
    }

    /**
     * Sets the node's real (db) id. A node's real id is the node's db row
     * id.
     *
     * @param id the node's db id
     */
    public void setRealId(int id) {
      realId = id;
    }
```

```
    /**
     * Returns the node’s real (db) id.
     *
     * @return the node’s db id
     */
    public int getRealId() {
      return realId;
    }

    /**
     * Returns the node’s type (db table TASKTYPE).
     *
     * @return type
     */
    public int getType() {
      return type;
    }

    /**
     * Sets the node’s type (db table TASKTYPE).
     *
     * @param i new type
     */
    public void setType(int i) {
      type = i;
    }

    /**
     * Returns the node’s description.
     *
     * @return description
     */
    public String getDescription() {
      return description;
    }

    /**
     * Sets the node’s description.
     *
     * @param s new description
     */
    public void setDescription(String s) {
      description = s;
    }

    /**
```

```
     * Returns the node's content.
     *
     * @return content
     */
    public String getContent() {
      return content;
    }

    /**
     * Sets the node's content.
     *
     * @param s new content
     */
    public void setContent(String s) {
      content = s;
    }

    /**
     * Returns property s using the BeanItem set in the constructor.
     *
     * @param s property to look for
     * @return property found; otherwise null
     */
    public Property getProperty(String s) {
      return bean.getItemProperty(s);
    }

    /**
     * Sets the node's state. State currently affects the node's border
     * color only (true = red, false = black).
     *
     * @param a state
     */
    public void setActive(boolean a) {
      active = a;
    }

    /**
     * Returns the node's x coordinate.
     *
     * @return the node's x coordinate
     */
    public int getX() {
      return x;
    }
```

```
    /**
     * Sets the node's x coordinate.
     *
     * @param x the node's new x coordinate
     */
    public void setX(int x) {
      this.x = x;
    }

    /**
     * Returns the node's y coordinate.
     *
     * @return the node's y coordinate
     */
    public int getY() {
      return y;
    }

    /**
     * Sets the node's y coordinate.
     *
     * @param y the node's y coordinate
     */
    public void setY(int y) {
      this.y = y;
    }

    /**
     * Adds a node to this node's children list. If the given node isn't
     * null or the node hasn't been defined earlier as this node's parent or
     * child, the node is added to the children list and this one is added
     * to the node's parents list.
     *
     * @param child node to be added
     */
    public void addChild(TreeNode child) {
      if (child == null || child == this || parents.contains(child) || children.contains(child)) {
        return;
      }

      children.add(child);
      child.addParent(this);
    }

    /**
     * Removes a node from this node's children list, also removing this
```

```
     * node from the given node's parents list.
     *
     * @param child node to be removed
     */
    public void removeChild(TreeNode child) {
      child.parents.remove(this);
      children.remove(child);
    }

    /**
     * Adds a node to this node's parents list.
     *
     * @param parent node to be added
     */
    public void addParent(TreeNode parent) {
      parents.add(parent);
    }

    /**
     * Removes a node from this node's parents list.
     *
     * @param parent node to be removed
     */
    public void removeParent(TreeNode parent) {
      parents.remove(parent);
    }

    /**
     * Removes all the connections between this node and the connected child
     * and parent nodes.
     */
    public void detach() {
      for (TreeNode child : children) {
        child.parents.remove(this);
      }

      for (TreeNode parent : parents) {
        parent.children.remove(this);
      }
    }

    /**
     * Returns the children list.
     *
     * @return children list
     */
```

```
    public LinkedList<TreeNode> getChildren() {
      return children;
    }

    /**
     * Returns the parents list.
     *
     * @return parents list
     */
    public LinkedList<TreeNode> getParents() {
      return parents;
    }

    /**
     * Moves the node to the given coordinates, its center positioned in
     * (x,y).
     */
    public void moveTo(int x, int y) {
      this.x = x - width / 2;
      this.y = y - height / 2;
    }

    /**
     * Draws the node to the current graphics context. The base of the node
     * is a solid rectangle filled with the defined base color, framed with
     * one pixel wide border (red if node active, black otherwise). Also, a
     * set of connector lines are drawn between this node and its children.
     *
     * @param g graphics context
     */
    public void paint(Graphics g) {
      g.setColor(baseColor);
      g.fillRect(x, y, NODE_WIDTH, NODE_HEIGHT);

      Color color;

      if (active) {
        color = Color.red;
        g.setColor(color);
        g.drawRect(x + 1, y + 1, NODE_WIDTH - 2, NODE_HEIGHT - 2);
      } else {
        color = Color.black;
        g.setColor(color);
      }
      g.drawRect(x, y, NODE_WIDTH, NODE_HEIGHT);
```

```
      g.setColor(Color.black);
      g.drawString(Integer.toString(id), x, y);

      g.setColor(CONNECTOR_COLOR);

      for (TreeNode child : children) {
        g.drawLine(x + NODE_WIDTH / 2, y + NODE_HEIGHT, child.x + NODE_WIDTH / 2, child.y);
      }
    }

    /**
     * Returns the node's string presentation.
     *
     * @return string presentation
     */
    @Override
    public String toString() {
      return Integer.toString(id);
    }
  }


  /**
   * SinglePathNode is a TreeNode that can have only one child. The amount of
   * parents is unlimited.
   */
  public class SinglePathNode extends TreeNode  implements Serializable {

    /**
     * SinglePathNode constructor.
     *
     * @param id node id
     * @param x the x coordinate
     * @param y the y coordinate
     */
    private SinglePathNode(int id, int x, int y) {
      super(id, x, y);
    }

    /**
     * Adds a node to this node's children list, removing any previously
     * existing nodes from the list.
     *
     * @param node node to be added
     */
    @Override
```

```
    public void addChild(TreeNode node) {
      if (children.size() > 0) {
        children.pop().parents.remove(this);
      }

      super.addChild(node);
    }
  }


  /**
   * MultiPathNode is a TreeNode that can have multiple children and multiple
   * exit paths all leading to different destinations (or the same if needed).
   */
  public class MultiPathNode extends TreeNode implements Serializable {

    // The list containing the created NodePaths.
    private LinkedList<NodePath> nodePaths;

    /**
     * MultiNodePath constructor.
     *
     * @param id node id
     * @param x the x coordinate
     * @param y the y coordinate
     */
    private MultiPathNode(int id, int x, int y) {
      super(id, x, y);
      nodePaths = new LinkedList<NodePath>();
    }

    /**
     * Creates an empty NodePath, adding it to the nodePaths list.
     */
    public void createNodePath() {
      nodePaths.add(new NodePath());
    }

    /**
     * Creates a nodepath with the given values and adds it to the paths
     * list.
     *
     * @param description the path's description
     * @param value the path's value
     * @param destinationNode the path's destination node
     */
```

```
    public void createNodePath(String description, int value, TreeNode destinationNode) {
      nodePaths.add(new NodePath(description, value, destinationNode));
    }

    /**
     * Returns the nodePaths list.
     *
     * @return nodePaths list
     */
    public LinkedList<NodePath> getNodePaths() {
      return nodePaths;
    }

    /**
     * Removes the given NodePath from the nodePaths list.
     *
     * @param nodePath nodePath to remove
     */
    public void removeNodePath(NodePath nodePath) {
      nodePaths.remove(nodePath);
    }

    /**
     * NodePath is used by MultiPathNode (actually RadioNode only) to define
     * different exit paths for the node. A NodePath could be say to act as
     * an instance of the database's table CHOICE, containing settings for
     * a single choice used in tasks that have multiple alternative paths
     * leading to the next tasks.
     */
    public class NodePath implements Serializable {

      // The node this choice leads to.
      private TreeNode destinationNode;
      // Amount of points choosing this choice gives.
      private int value;
      // Description displayed next to this choice.
      private String description;
      // BeanItem used in accessing the path's values through SettingsView
      private BeanItem bean;

      /**
       * NodePath default constructor.
       */
      private NodePath() {
        value = -1;
        description = "";
```

```
    bean = new BeanItem(this);
  }

  /**
   * NodePath constructor for pre-defined values.
   *
   * @param description the path's description
   * @param value the path's value
   * @param destinationNode the path's destination node
   */
  private NodePath(String description, int value, TreeNode destinationNode) {
    this.description = description;
    this.value = value;
    this.destinationNode = destinationNode;
    bean = new BeanItem(this);
  }

  /**
   * Sets the destination node. Used by BeanItem only.
   *
   * @param node new destination node
   */
  public void setDestinationNode(TreeNode node) {
    destinationNode = node;
  }

  /**
   * Returns the destination node. Used by BeanItem only.
   *
   * @return destination node
   */
  public TreeNode getDestinationNode() {
    return destinationNode;
  }

  /**
   * Sets the value (points) given. Used by BeanItem only.
   *
   * @param v new value
   */
  public void setValue(int v) {
    value = v;
  }

  /**
   * Returns the value (points) given. Used by BeanItem only.
```

```
       *
       * @return value
       */
      public int getValue() {
        return value;
      }

      /**
       * Sets the description. Used by BeanItem only.
       *
       * @param s new description
       */
      public void setDescription(String s) {
        description = s;
      }

      /**
       * Returns the description. Used by BeanItem only.
       *
       * @return description
       */
      public String getDescription() {
        return description;
      }

      /**
       * Returns property s using the BeanItem set in the constructor.
       *
       * @param p property to look for
       * @return property found; otherwise null
       */
      public Property getProperty(String p) {
        return bean.getItemProperty(p);
      }
    }
  }


  /**
   * RootNode is an unremoveable node that has only one child and cannot have
   * any parents.
   */
  public class RootNode extends SinglePathNode implements Serializable {

    /**
     * RootNode constructor.
```

```
     *
     * @param id the node's id
     * @param x the node's x coordinate
     * @param y the node's y coordinate
     */
    private RootNode(int id, int x, int y) {
      super(id, x, y);
      super.baseColor = Color.white;
      super.setDescription("Aloitus");
      super.setType(1);
    }
  }


  /**
   * TextNode is a node that contains plain text. A TextNode can have several
   * parents but only one or zero children.
   */
  private class TextNode extends SinglePathNode implements Serializable {

    /**
     * TextNode constructor.
     *
     * @param id the node's id
     * @param x the node's x coordinate
     * @param y the node's y coordinate
     */
    private TextNode(int id, int x, int y) {
      super(id, x, y);
      super.baseColor = Color.cyan;
      super.setDescription("Tekstitehtävä");
      super.setType(1);
    }
  }


  /**
   * RadioNode is a node that contains paths leading to different (or just
   * one) TreeNode(s). Like TextNode, a RadioNode can have several parents,
   * but it also can have several children, too. It is also possible for a
   * RadioNode to have just one child with several paths leading to it.
   */
  protected class RadioNode extends MultiPathNode implements Serializable {

    /**
     * RadioNode constructor.
```

```
     *
     * @param id the node’s id
     * @param x the node’s x coordinate
     * @param y the node’s y coordinate
     */
    private RadioNode(int id, int x, int y) {
      super(id, x, y);
      super.baseColor = Color.orange;
      super.setDescription("Valintatehtävä");
      super.setType(2);
    }
  }
}
```