

Graph, Tree and DAG Object and Window Structures

In the previous chapter, we learnt how to create complex logical structures included structured and composite objects and shapes. In this chapter, we will classify these structured into three important categories of increasing complexity: trees, DAGS, and graphs. In later courses, you will theoretically analyze these structured in great depth. Here, we will see some of the practical consequences, of (deliberately or accidentally) choosing among these categories. We will redo the Cartesian plane example to illustrate the structures and consequences.

Cartesian Plane with Composite Line Objects

Recall that we have created two different interfaces for a geometric line. One, called `Line`, was both an atomic shape and atomic object – all properties were primitive.

```
public interface Line {  
    public int getX();  
    public void setX(int newX);  
    public int getY();  
    public void setY(int newY);  
    ...  
}
```

The other, `LineWithObjectProperty`, was also an atomic shape but a composite object – its location was an object.

```
public interface LineWithObjectProperty {  
    public Point getLocation();  
    public void setLocation(Point newLocation);  
    ...  
}
```

In our previous `CartesianPlane` solution, we embedded two `Line` objects to represent the two axes:

```
public interface CartesianPlane {  
    public Line getXAxis();
```

```

    public Line getYAxis();
    ...
}

```

Let us make the problem more interesting by representing the axes as `LineWithObjectProperty` and also making the locations of these two axes as properties of the plane.

```

    public interface DAGCartesianPlane {
        public LineWithObjectProperty getXAxis();
        public LineWithObjectProperty getYAxis();
        public Point getXAxisLocation();
        public Point getYAxisLocation();
        ...
    }

```

In the implementation of this interface, we will now create two additional instance variables to represent the two additional `Point` properties, and use `LineWithObjectProperty` as the type of the two axes:

```

LineWithObjectProperty xAxis, yAxis;
Point xAxisLocation, yAxisLocation;

```

As before, we will write functions to compute the location of the axes from the axes length. In the previous solution, for each axis, we wrote two functions, one of the X coordinate, and one for the Y coordinate. As our location is now a single object, we can now write a single function, for each axis, that computes both coordinates.

```

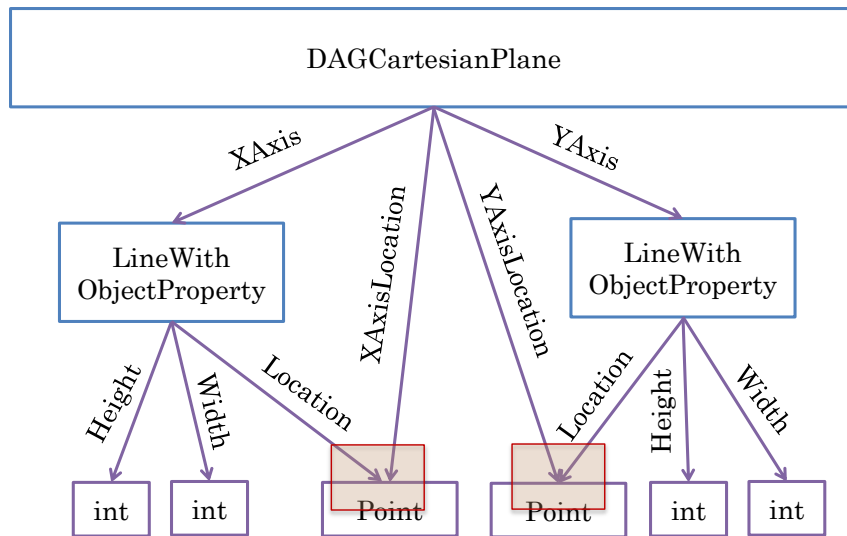
xAxisLocation = toXAxisLocation();
yAxisLocation = toYAxisLocation();
xAxis.setLocation(xAxisLocation );
yAxis.setLocation(yAxisLocation );

```

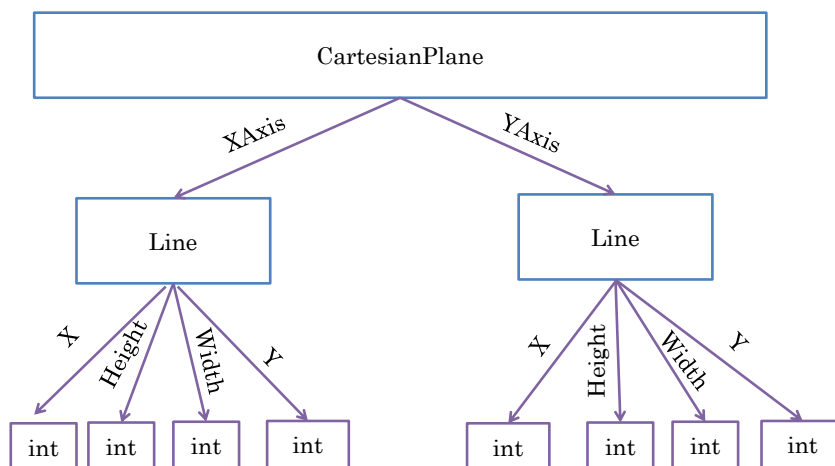
Let us try to understand the effect of the `setLocation()` calls. As we saw earlier, when we assign an object variable to another, the address in the former is assigned to the latter. That means, after the assignment, both variables point to the same object.

Direct Acyclic Graph (DAG Structure), Graph and Tree

The result of the assignments above is the logical structure shown in Figure ???.



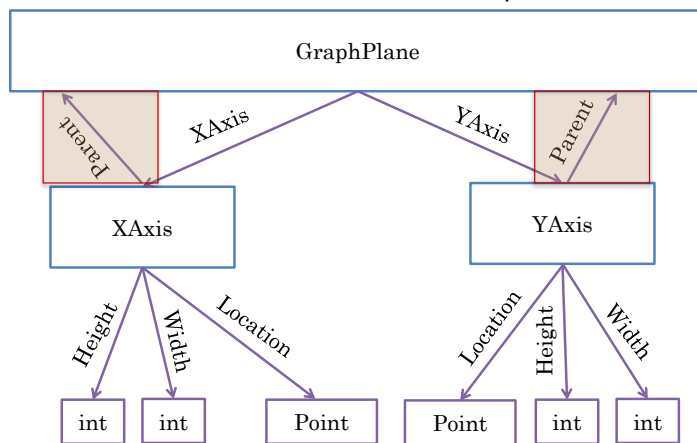
This structure is different from the ones we have seen so far. In previous logical and physical structures, each non-root node had exactly one parent, that is, exactly one property or variable pointing to it. This is illustrated in the logical structure of the previous version of the Cartesian plane.



In the structure of the new version, the locations of the two axes have two properties referencing them. The location of the X axis is referenced both by the (a) Location property of the XAxis and (b) the XAxisLocation property of the Cartesian plane. Similarly, the location of the Y axis has two parents. As a result, there are multiple paths to the two location objects – one directly from the Cartesian plane and one through the two axes.

We can make the logical structure even more interesting by having each axis have a property, **Parent**, that refers to the Cartesian plane that contains it. Recall from the previous chapter that such a reference allows the creation of specialized XAxis and YAxis that compute their dependent locations based on the

independent XAxis length property of the parent Cartesian Plane object. Figure ??? shows the logical structure of the new version of Cartesian plane.



Given our terminology from the previous chapter, the two axes are “parents” of the Cartesian plane as they have properties pointing to them. As a result, we have a cyclic structure, in which, if we follow the XAxis edge from the Cartesian plane to the XAxis and then Parent edge from the XAxis, we return back to the Cartesian plane. Similarly, the edge to YAxis leads to a cycle.

These three examples define three classes of node-edge structures.

Tree: Each node has a single incoming edge, that is, a single parent. As a result it is not possible to have multiple paths to a node, which in turn, precludes cycles. Each internal node roots a subtree in which no node has an edge to a node in any other subtree

Directed Acyclic Graph (DAG): This is a generalization of a tree in which it is possible to have multiple incoming edges to a node, but there is no return path to a node, that is, there is no cycle.

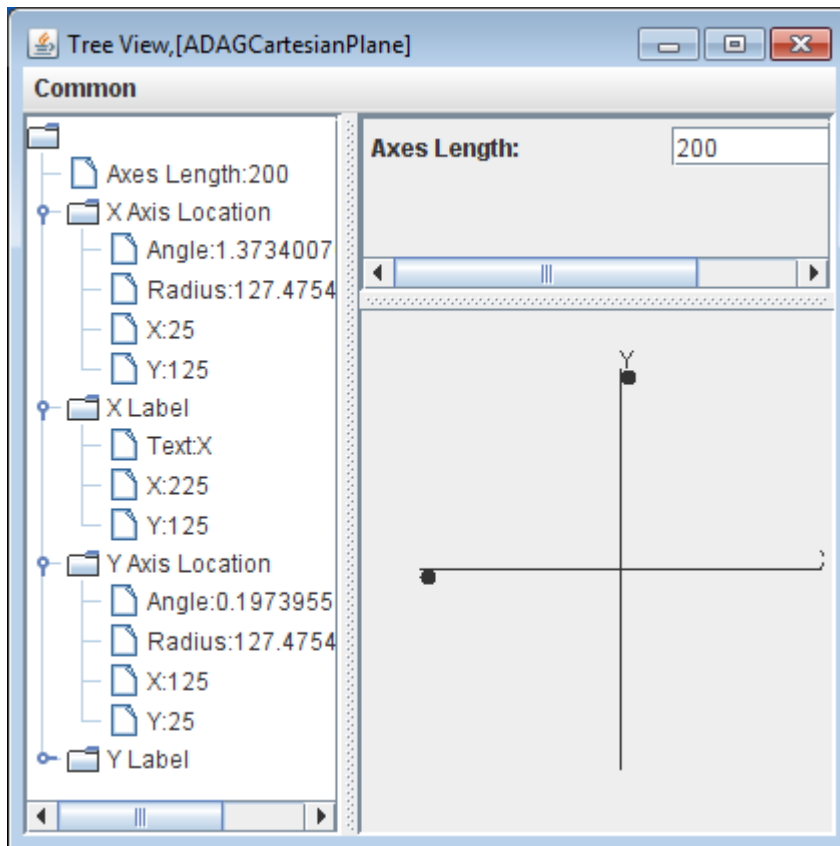
Graph: This is the most general structure in which any node can be connected to any other node. Thus, it is possible to have cycles and, of course, multiple paths to a node.

One would think that a more flexible structure is superior. The consequence of flexibility is complexity, which makes it harder to reason and process the structure in many situations. For this reason, many useful algorithms can handle only trees,

Converting a DAG to a Tree

One such algorithm is one that displays a logical structure in a tree view. As its name suggests, such a view is designed for displaying trees. The reason is that it is a pure hierarchical view in which the display of a child node is nested under the display of the parent node. If a node has multiple parents, under which one of these parents should it be nested? Thus, when given a DAG to display, ObjectEditor automatically removes certain nodes from the DAG to convert it to a tree. This is shown in Figure ???, which is a display of the DAG Cartesian plane. Neither the X Axis nor the Y axis is displayed in the tree view, as these two nodes have been removed by ObjectEditor for this view.

These two nodes are displayed in the graphics view, as it does not show a hierarchical structure, and simply displays all atomic shapes at the positions specified by their coordinates. However, the graphical view has its own problems. The location property of each axis follows the pattern for the Point shape. As a result, this shape is displayed in the graphics view, which is good for debugging purposes but not the ultimate end-user display.



If we really wanted to create a DAG logical structure that is displayed in the different ObjectEditor views, two things went wrong in our solution. First, ObjectEditor decided which nodes should be removed from the DAG to display the tree view. Second, the two location objects were needlessly displayed in the graphics view. Both problems could have been avoided by asking ObjectEditor to remove the location properties of the axes from the *logical structure* to create the *displayed logical structure*. The displayed logical structure is the structure displayed by ObjectEditor and is specified by labelling certain nodes in the logical structure as invisible.

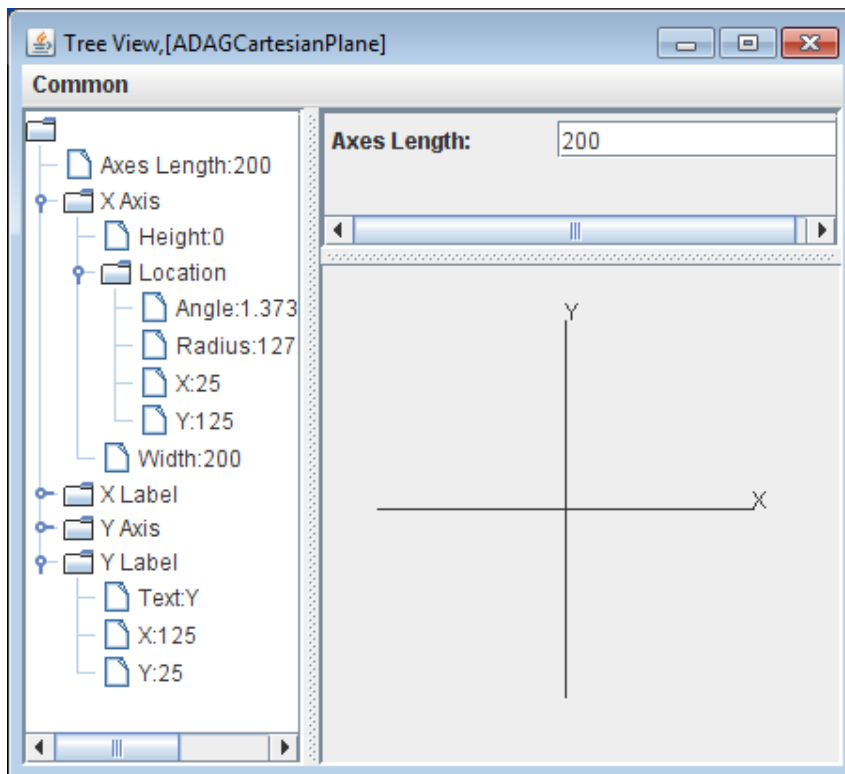
We will use annotations to mark invisible nodes.

```
public class ADAGCartesianPlane implements DAGCartesianPlane {
    @Visible(false)
    public Point getXAxisLocation() {
        return xAxisLocation;
    }
}
```

```
@Visible(false)
public Point getYAxisLocation() {
    return yAxisLocation;
}
```

....

If a getter of a property P is associated with the `@Visible(false)` annotation, then that property is not considered part of the displayed logical structure. Thus, the annotations above ensure that the `XAxisLocation` and `YAxisLocation` properties are not considered part of the displayed logical structure. With these transformations, the two axes reappear in the logical structure, and the Point location objects disappear from both views, as shown in Figure ???.



In summary, a tree view can handle only tree structures. If given a non-tree structure, ObjectEditor chooses which nodes to remove to create a tree view, and thus, may remove the wrong nodes. `@Visible` annotations can be used by the programmers to determine which nodes are removed from the logical structure to determine the displayed structure.

Window Structure

A tree view, by its very name, declares itself incapable of handling non -trees. Perhaps more surprising, the GUIs we create, such as in the ObjectEditor main view, are also capable of handling only tree structures. Let us take this opportunity to introduce both the nature of these GUIs and the limitations of the associated structures.

These GUIs consist of a bunch of nested windows. A window is a rectangular region of the screen in which one can display text or graphics. A top-level window is one we can move and resize using the capabilities of the underlying operating system. Non top-level or internal windows are nested inside the top-level windows. A software system that supports the creation of different kinds of top-level and non top-level windows is called a (user-interface) toolkit. Like a physical toolkit, it allows a person to assemble different kinds of windows to create a GUI. Java originally provided the AWT toolkit, which exposed the top-level and non top-level operating system windows to the programmer. Later, it added the Swing toolkit, which provided a Java-specific implementation of the non top-level windows. The names of the Swing classes were derived by adding the prefix J to the names of the corresponding AWT windows. Non top-level windows, which contain specialized behavior such as scrolled text and buttons, are also called “widgets”. This bland unspecific term suggests that there is no constraint on the behavior of these windows. They do have some common behavior. When we study inheritance, we will see that this behavior is captured by Java class, Component. Thus, a component is another term for a Java widget.

In AWT and Swing, a top-level window is instance of the Frame or JFrame class. The other windows are instances of different kinds of classes, depending on what they display. The following code illustrates some of the Swing classes and their use. This code creates the GUI of Figure ???.

```
public static JFrame createTree {  
    JFrame frame = new JFrame();  
    JSplitPane splitPane = new JSplitPane();  
    frame.add(splitPane);  
    JPanel leftPanel = new JPanel();  
    JPanel rightPanel = new JPanel();  
    splitPane.setLeftComponent(leftPanel);  
    splitPane.setRightComponent(rightPanel);  
    JTextField textField = new JTextField("Edit me");  
    leftPanel.add(textField);  
    JButton button = new JButton ("Press me");  
    rightPanel.add(button);  
    frame.setSize(200, 100);  
    frame.setVisible(true);  
    return frame;  
}
```



The top-level window of the GUI is created by instantiating the class JFrame.

```
JFrame frame = new JFrame();
```

We now want to divide or split this window into two components. This is done by creating an instance of JSplitPane:

```
JSplitPane splitPane = new JSplitPane();
```

and “adding” it to the JFrame instance:

```
frame.add(splitPane);
```

The add call is the most interesting one from the point of view of this chapter. It creates an edge between the JFrame instance and the JSplitPane, making the latter the child of the former. A child window is always displayed within its parent window - any contents outside the parent window is clipped, that is, not displayed.

The JSplitPane splits its parent window into a left and right component. The code next creates two instances of JPanel and makes them the children of these two components, respectively by making setter calls:

```
JPanel leftPanel = new JPanel();
```

```
JPanel rightPanel = new JPanel();
```

```
splitPane.setLeftComponent(leftPanel);
```

```
splitPane.setRightComponent(rightPanel);
```

A JPanel is a container or internal node in the window structure to which we can add an arbitrary number of other windows. In our example, we add only one window to each of the two panels. To the leftPanel, we add an instance of JTextField and to the rightPanel, we add an instance of JButton.

```
JTextField textField = new JTextField("Edit me");
```

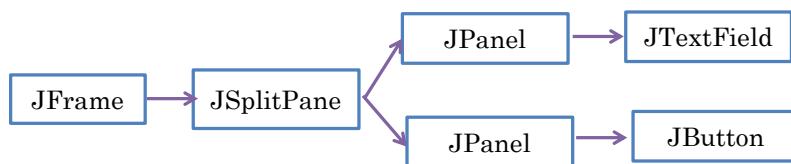
```
leftPanel.add(textField);
```

```
JButton button = new JButton("Press me");
```

```
rightPanel.add(button);
```

A JTextField displays text. Its constructor takes as a parameter the String to be displayed. A JButton displays a labelled button; its constructor takes the label as an argument.

These calls, together, create the following tree structure:



As we saw before, this structure is created incrementally, by adding one node at a time to the tree. If it was displayed during its construction, the GUI would flicker. Therefore, the Swing toolkit provides a special setVisible() call, which can be made after the construction of the structure is complete.

```
frame.setVisible(true);
```

This call displays the structure. Before making this call, we must set the size of the top window – the JFrame instance.

```
frame.setSize(200, 100);
```

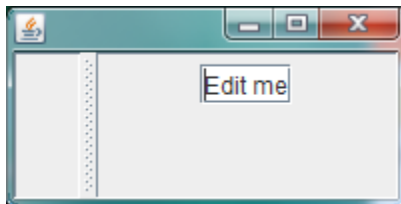
Any content in descendent windows that does not fit in the top-level window is, of course, clipped. Some widgets adjust their size to fit their contents within the boundaries of their parents.

Tree, DAG and Graph GUI Structure

The window structure we created in the example above is a tree, with each non-root node having exactly one root. The Java API allows us to make any node a child of any other node. Let us use this API to try and create a DAG structure by making the following two calls;

```
leftPanel.add(textField);  
rightPanel.add(textField);
```

Here, we have made both components of the split pane parents of a single text field. What should Java do with these calls? Java's algorithm for clipping and displaying windows, like a tree algorithm, requires a child windows' contents to be displayed within the parents' contents. If a child has multiple parents, within which parent should it be displayed? For this reason, Java converts our DAG to a tree by essentially re-parenting a child each time we set its parent. In other words, the last call to set the parent "wins", that is, takes precedence. In the example above, the second add call makes the right panel essentially adopt the text field, and the left panel is no longer the parent. As a result, we get the display of Figure ???, in which the left panel is childless and the right panel has the text field as a child.



Let us now try and create a cycle in the window structure by making the following calls:

```
splitPane.setRightComponent(rightPanel);  
rightPanel.add(splitPane)
```

Here, we first make the right panel a child panel of the split pane and then do the reverse. This time, Java does take the "the last call wins" approach and simply gives an error message saying that a cycle is being created.

```
Exception in thread "main" java.lang.IllegalArgumentException: adding container's parent to itself  
    at java.awt.Container.checkAddToSelf(Unknown Source)  
    at java.awt.Container.addImpl(Unknown Source)  
    at java.awt.Container.add(Unknown Source)  
    at lectures.composite.TreeVsDAG_Objects_Windows.WindowGraphCreator.main(WindowGraphCreator.
```

In summary, a window structure is also a tree structure. Though the Java API seems to allow the creation of DAG and graph structures, in fact, these calls re-parent a child when we “add” a new parent and give an error when we create a cycle.

Declarative and Procedural Structure Specifications

We have seen three ways of specifying node-edge structures. Physical structures of objects are created by *declaring* instance variables. Logical structures of objects are created by *declaring* getters and possibly setters. Window structures are created by making special API calls. A specification made by making method calls is called a *procedural specification*. The alternative is a *declarative specification* made using some syntax established in some language. The language may be a programming language such as Java or some other kind of language such as XML (eXtensible Markup Language) about which you might have heard and will probably learn in a Web course. As physical and logical structures are specified using Java variable and method declarations, they are specified declaratively. Window structures are specified procedurally using Java AWT/ Swing calls.

Translating between Structures

There is another way to distinguish between logical/physical structures and window structures. ObjectEditor and a (visual) debugger display the logical and physical structures, respectively, by mapping these to window structures. For instance, in the main view, ObjectEditor maps Strings and primitive property to text fields and other non-shape properties to panels. One can, thus, think of it as a translator between logical and window structures, and a (visual) debugger as a translator between physical and window structures. Both ObjectEditor and debuggers are, of course, far more than translators. Moreover, not all debuggers create window structures to display logical structures – textual debuggers display them in a console window.

More on Toolkits

Similarly, toolkits are more than an API to create window structures. Later, we will see we can change the “output” displayed in a widget and receive “input” from it.