Glazed Lists

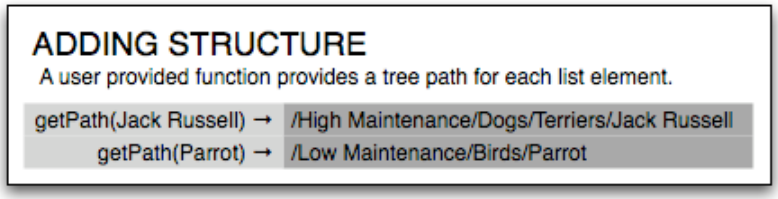
Glazed Lists Internals: TreeList and Barcode, Part 1

Jesse Wilson, jesse@swank.ca

November 3, 2006

In this paper I discuss some of the interesting internals of Glazed Lists' TreeList class, to provide algorithm candy to the interested developer. We start with a source EventList. In this case, it's a list of pets.

To apply the TreeList transformation to the source list, a user-provided function specifies a tree path for each element in source. In this example, the element `Budgie` is mapped the path `/Low Maintenance/Birds/Budgie`



The TreeList ensures there are list elements for all parts of the tree path. Path elements that don't appear in the source list have list elements created for them in the TreeList. These elements are considered 'virtual' elements in the TreeList because they have no counterparts in the source list. In the diagram, these virtual elements are all green.

One consequence of having virtual elements is that when a source element is deleted (such as `Goldfish`), it may leave behind a parent that has no children left (ie. `Fish`). When this happens, TreeList automatically deletes the obsolete virtual parent element. Similarly, inserted elements may cause TreeList to create the required virtual parents.

In order to respond to changes, TreeList must be able to map between indices in itself (`0` thru `10`) and the indices in the source list (`0` thru `4`). For example, if the user calls `treeList.remove(8)` the TreeList handles this by mapping index `8` in itself to index `3` in the source list, then calling `sourceList.remove(3)`. Similarly, if TreeList receives notification that the source list has been updated at index `1`, TreeList will map this index and notify its listeners that TreeList index `4` has been updated. Although they use different indices in the different lists, the updated value is `Welsh Terrier` in both cases.

This mapping is implemented using our custom ADT "Barcode" that manages sets of indices – *blue* (real) indices and *green* (virtual) indices. There's also an implicit set of indices – the complete set of indices, which we call *overall* indices. Once we tell Barcode that indices `3`, `4`, `7`, `8` and `10` are blue and the rest are green, it can answer our questions:

- what's the overall index of the first blue element?
- what's the green index of the element at overall index `9`?
- what color is the element at index `6`

Barcode is dynamic, so we can tell it that `5` blue elements have been inserted at overall index `8` and it will adjust the mapping.

Barcode is a useful ADT because it manages index-mapping state in a general way. In FilterList, it conveniently manages both filter state and index mapping. In UniqueList and GroupingList, Barcode manages the indices of duplicate elements.

Glazed Lists

Glazed Lists Internals: TreeList and Barcode, Part 2

Jesse Wilson, jesse@swank.ca

November 6, 2006

In the first part, I showed how Glazed Lists' Barcode ADT is used to provide mappings for a subset of a list's indices. In this paper, we expand on those concepts.

We have already distinguished between nodes that are *real* (shown in blue) and virtual (shown in *green*). A value is real if it exists in the source EventList, or virtual if it was created by TreeList to provide a missing parent for the tree structure. This is the *real/virtual* state of a node.

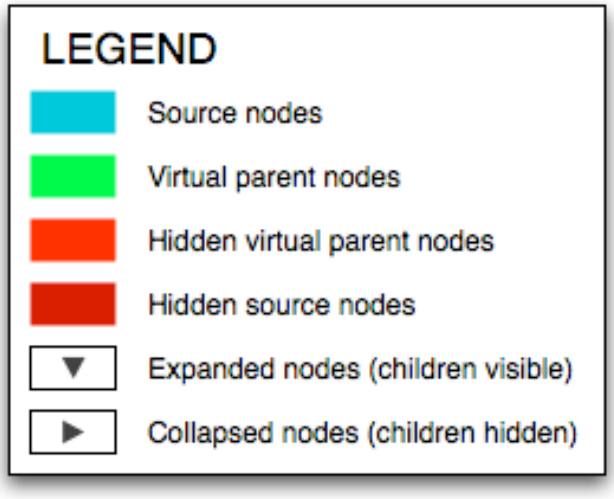
Nodes in trees can be expanded or collapsed. Consider the tree views in Windows Explorer and the Mac OS X Finder. Folder nodes can be collapsed, hiding everything within. When they're expanded, the contained files become visible. Note that a collapsed node is itself visible – it is the child nodes that become hidden. Therefore, we have two new states for each node: the *expanded/collapsed* state and the *visible/hidden* state.

A node is expanded or collapsed directly. To modify this state on a node in the user interface, locate the node and click to toggle its expand/collapse icon. The *hidden/visible* state is less direct. To hide a node, collapse any of its ancestor nodes such as its direct parent or the root node. Conversely to make a node visible, expand all of its ancestor nodes.

TreeList manages *hidden/visible* using a same pattern described in the first paper for *real/virtual* but with a different set of colors. For this example, hidden nodes are red and visible nodes are aqua. When a user requests the value at *visible* index `2`, `Low Maintenance`, TreeList maps this to index `5` in the *overall* tree. The reverse mapping is also used: if the *overall* index `8`, `Parrot`, is updated then TreeList notifies its listeners about a change at *visible* index `5`.

The final piece of this puzzle is combining hidden/visible filtering with real/virtual data injection. Combining our two pairs of states, a node is either real+visible, real+hidden, virtual+visible or virtual+hidden.

Suppose the source element `Budgie` is updated. TreeList will map its source index `3` to overall index `7` because four virtual elements precede it. Then the overall index `7` is mapped to visible index `4` because three hidden elements precede that. Finally, TreeList notifies its listeners about the update at visible index `4`.



A special version of the Barcode ADT performs this index conversion in a single step – given a source index it produces the related visible index or vice versa.

The Barcode ADT is the foundation of TreeList, making it possible to track both virtual nodes and collapsed nodes without sacrificing simplicity or performance.