TreeView Widget

User's Guide & Cookbook

The purpose of this document is to give you a comprehensive overview of the features and facilities provided by *TreeView Widgets* and how to access them programmatically. It starts with an introductory chapter. Subsequently step-by-step instructions show how to build applications employing new data set widgets.

Contents

Introduction

The classes in this package are designed to provide the functionality of TreeView widgets. It enables you to enrich your application with facilities to display and manage hierarchical structures in a consistent and familiar way. TreeView widget adopts the Look and Feel of Tree Controls known from Windows 95 applications, such as explorer.

You can manipulate the appearance of a TreeView in various ways: You can enable or disable each, the use of icons, lines or expand-collapse buttons ([+], [-]). All this is done within the TreeView's properties page. Furthermore you can either use pre defined icons or prepare a TreeView to use your own application specific icons.

User Interface

The user interface of a TreeView widget allows a user to browse through a hierarchy by means of selecting and expanding or collapsing entries. A hierarchy displayed in a TreeView may have one or more top level entries comprising the roots. Each subordinate entry of a root is indented by an offset according to its level within the hierarchy.

TreeView inherits its standard behavior from SequenceView. Thus all the user interface details of list boxes (scrolling, target selection by typing in leading characters, etc.) are available in TreeView too. Additionally you can expand/collapse entries either by mouse or by keyboard shortcuts.

You can toggle the expand/collapse status of an entry by one of these:

- Double Click on an entry
- Single Click on an entry's expand-collapse button ([+], [-]) (doesn't change the selection)
- Pressing Return / Enter
- Pressing Cursor Left / Right (if the actually selected entry is already collapsed / expanded, selection is moved to its parent / first child)

Further keyboard shortcuts are:

- Up / Down moves the selection to the previous/next entry
- Ctrl Up / Ctrl Down moves the selection to the previous/next entry at the same level of hierarchy, skipping children entries on lower levels.

- Ctrl Left moves the selection to the currently selected entry’s parent entry.
- Page Up/Page Down moves the selection one page up/down

If you hold down the Shift key while expanding or collapsing an entry (both, either by mouse or by keyboard), the complete subtree starting at that entry will be expanded or collapsed.

Programming Interface Concepts

From a programmer's point of view, a TreeView widget behaves very much like a SequenceView. In fact class *TreeView* is derived from *SequenceView* and inherits most of its functionality — and hence its programming interface — from its superclass. As with SequenceViews, instances of *SelectionInList* are used as models to a TreeView.

The more specific concepts of the TreeView programming interface are described in the following. This is a brief summary of these concepts:

- Tree adaptors and children blocks
- Tree nodes and node subjects
- Manufacturing of tree nodes
- Fetching child entries
- Equality vs. identity comparison of tree nodes
- Single- vs. multiple-parent hierarchies

Tree adaptors and children blocks

The subject (list) of a SelectionInList that is used as model of a TreeView, is an instance of class **TreeAdaptor**. This class basically provides the means to express and adapt to arbitrary hierarchical relationships in a uniform way; i.e. you might use a TreeAdaptor to adapt your tree view widget to the directory hierarchy of your file system, or as well configure it to be used to navigate through the VisualWorks class hierarchy.

In principle, you configure a TreeAdaptor with an object comprising the root(s) of the hierarchy and a so called **children block** that is responsible for retrieving and returning a collection of subordinate hierarchy entries or nodes (the children) to a given node (the parent). This children block will be evaluated by the TreeView/TreeAdaptor automatically whenever the children of a certain entry need to be fetched for the first time, such as on expanding a node. The following examples illustrate the use of TreeAdaptors and children blocks:

```
TreeAdaptor new
        childrenBlock [:aClass | aClass subclasses];
        root: Object

TreeAdaptor new
        childrenBlock: [:aDirectory | aDirectory asFilename directoryContents];
        root: Object
```

TreeAdaptor inherits from *SequencableCollection* and thus provides you with the familiar enumeration and accessing protocols used with collections. Not least, this makes it possible for instances of TreeAdaptor to be used as subjects to a SelectionInList.

Tree nodes and node subjects

Each domain model object to be shown in a TreeView is finally wrapped in an instance of class *TreeNode*. The object wrapped by a TreeNode is called a tree node's **subject**. A TreeNode keeps track of an entry's expand/collapse status and caches the children once fetched by a children block. It also provides various options to additionally describe an entry. These options allow to specify:

- The type of node (is used to determine the icon to display).
- A specific icon for this node (overrides the default icon detection by type).
- A specific display string to use as the entry's label.
- Information about whether children can be fetched or not.

Automatic vs. explicit manufacturing of tree nodes

When returning child entries in a children block, the collection returned can contain either instances of domain model entity classes or tree nodes wrapping the domain model object in arbitrary combination. In case of the latter, the tree adaptor will manufacture appropriate tree nodes automatically, one for each hierarchy entry. Automatically manufactured tree nodes will have default property values (type = *#folder*, the subjects *displayString*, etc.).

Instead of having the tree adaptor manufacture the *TreeNode* instances automatically, you may decide to provide for explicit manufacturing of tree nodes in your application and return collections of readily configured *TreeNode* instances from your children block. This gives you more control over the kind of tree nodes to be used (e.g. instances of *IndentityTreeNode* or specialized subclasses) and the node properties to be set.

Fetching children on demand vs. pre-fetching children

As mentioned above, a tree adaptor’s children block is used to automatically fetch a node’s child entries on demand, such as on the first expansion of a node. Thus, once having provided a tree adaptor with an appropriate children block, you will never have to explicitly fetch a node’s children in your application.

However, there may arise situations in which you would want to pre-fetch a node’s children. An example is an application that doesn’t know whether to display an expand button for a node or not, unless the node’s children have been fetched. In this case you can arrange for pre-fetching a node’s children at the moment it is returned from a children block.

Equality vs. identity comparison of TreeNodes

There are several occasions for a TreeView to look up a tree node for a given domain model object, i.e. to fetch the very instance of *TreeNode* that has a certain domain model object as its subject. To detect the tree node in question, each registered node’s subject has to be compared to the specified domain model object. This comparison can be based on either equality or on identity of subjects.

The default, as being implemented in class *TreeNode*, arranges for equality comparison. Identity comparison is supported in class *IdentityTreeNode*, a subclass of *TreeNode*. On principle, identity comparison results in a faster processing and behaviour, but may not be appropriate in each case. Equality comparison, on the other hand, is less efficient but more general in will thus work in each case.

Single- vs. multiple-parent hierarchies

Principally, tree views are designed to display single-parent hierarchies, i.e. each entry in an hierarchy is expected to have exactly one and only one parent node. In consequence a *TreeNode* keeps a reference to **exactly one parent** node and each domain model hierarchy entry is represented in a TreeView by **exactly one instance** of *TreeNode*.

The basic difficulty when dealing with multiple-parent hierarchies in a TreeView is, that a certain entry may be displayed several times in the TreeView. This is because if a node has more than one parent, it would have to be displayed in each of these parents’ children branches. In result, there may be **more than one instances** of *TreeNode* for a certain domain model hierarchy entry.

There is some basic-level support for adapting to multiple-parent hierarchies through class *MultipleParentTreeAdaptor* a subclass of *TreeAdaptor*. This class cares for correct detection and manipulation of each instance of

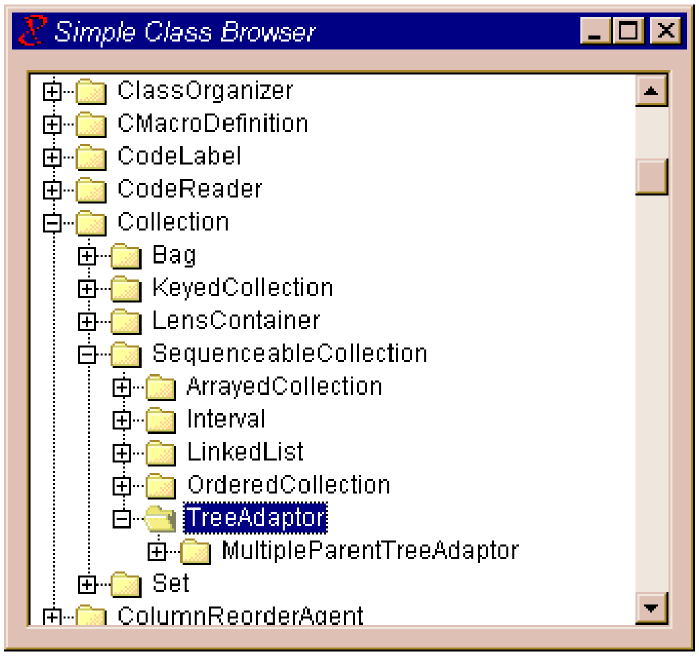
TreeNode for a given domain model object in the common operations, such as on adding or removing nodes.

Summary of Features

- TreeView enriches your applications with capabilities to display and manage varying hierarchical structures in a consistent and familiar way,
- Allows you to affect the appearance through the use of icons, lines, and expand-collapse buttons.
- Adopts the Look and Feel of Windows 95 tree controls,
- Inherits most of its programming interface from SequenceView.
- Allows to express and adapt to arbitrary forms of hierarchies through the concept of tree adaptors,
- Provides automatic manufacturing of tree nodes and fetching of child entries on demand.

Tutorial Example 1: A Simple Class Hierarchy Browser

This example shows the basic steps to employ a TreeView widget in a VisualWorks application. As an example application, we shall implement a simple class browser which uses a TreeView to display the inheritance hierarchy of the classes being present in your image. It will be shown how a tree view is added to a canvas, how entries can be expanded and collapsed programmatically and how entries can be added and removed dynamically.



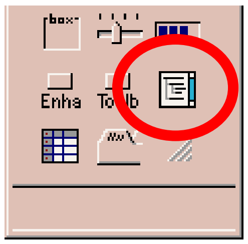
Adding a TreeView

Strategy

Since tree view widget inherits from list widget, it depends on two value models, a selection holder and a selection index holder, supplied through an instance of *SelectionInList*. Unlike a list box however, instead of a *List*, a tree view expects an instance of *TreeAdaptor* as the list in the SelectionInList in-stance.

TreeAdaptors provide the means to express and adapt to varying hierarchical relationships in a uniform way. Basically a TreeAdaptor is configured with an object comprising the root(s) of the hierarchy and a block — the so called *childrenBlock* — that is responsible for retrieving and returning a collection of subordinate hierarchy entries or nodes (the children) to a given node (the parent).

A TreeAdaptor wraps all entries returned by the children block in instances of *TreeNode*. Basically, a TreeNode keeps track of an entry's expand/collapse status. It is also responsible for providing a string to display as an entry's label. As a default the entry's *displayString* method is used for this purpose.

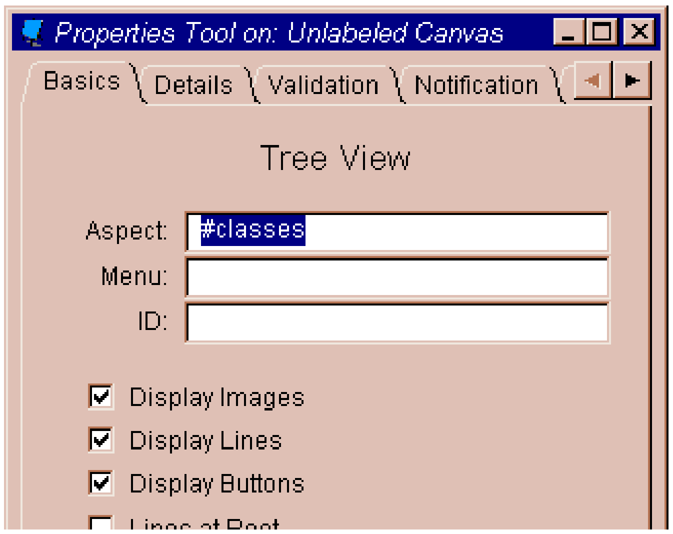


Basic Steps

Tutorial Example: *SimpleClassBrowser*

1. Use a Palette to add a TreeView widget to the canvas. (The TreeView placed on the canvas will show some ex-ample hierarchy to give you a preview of how it will look like. The contents of this example hierarchy will always remain the same.)

2. In the Properties Tool's Basics Page, fill in the TreeView's **Aspect** property with the name of the method that will return an instance of *SelectionInList*. (In this example we'll call it *classes*)



Properties Tool on: Unlabeled Canvas

Basics | Details | Validation | Notification

Tree View

Aspect: #classes
Menu:
ID:

[✓] Display Images
[✓] Display Lines
[✓] Display Buttons
[ ] Lines at Root

(You may also want to try out different modes of appearance by checking or un-checking the „Display Images“, „Display Lines“ and „Display Buttons“ check boxes.)

3. Use the canvas’s **define** command or a System Browser to add an in-stance variable (*classes*) to the application model to hold the *Se-lec-tionInList*.

4. Use the canvas’s **define** command or a System Browser to create the aspect method you named in step 2 (*classes*).

```
classes
    ^classes
```

5. Use a System Browser to initialize the instance variable you created in step 3 (*classes*), usually in an *initialize* method. You initialize the variable with an instance of *SelectionInList* that is itself initialized with an instance of *TreeAdaptor*:

```
initialize
    super initialize.
    classes := SelectionInList with:
        (TreeAdaptor new
            childrenBlock: [:aClass | aClass subclasses];
            root: Object)
```

Variant A: Expanding the root node

You may want your class inheritance browser to start up not just showing a single entry initially, but also the first level of children to that entry. You can achieve this by initially expanding the root entry programmatically. Use one of the TreeAdaptor’s *expand:* messages to do so:

```
initialize

    super initialize.
    classes := SelectionInList with:
        (TreeAdaptor new
            childrenBlock: [:aClass | aClass subclasses];
            root: Object).
    classes list expand: classes list rootNode.
```

You could as well use other variants of the *expand:* messages provided by TreeAdaptor class, such as:

```
    classes list expand: Object.
```

This will cause the TreeAdaptor to search for the TreeNode instance for entry Object (it does so by using equality comparison) and then expanding that node. This would correspond to:

```
    classes list expand: (classes list nodeFor: Object).
```

You can also access (and thus expand or collapse) entries by index:

```
    classes list expandAt: 1.
```

Variant B: Using more than one root

Instead of starting the display of the classes' inheritance hierarchy at class Object you can insert any other class in place of Object in basic step 5. You can also provide more than one root entry by sending a collection of root objects to the TreeAdaptor:

```
initialize

    | roots |
    super initialize.
    roots := #(Model View Controller) collect: [:each | Smalltalk at: each].
    classes := SelectionInList with:
            (TreeAdaptor new
                    childrenBlock: [:aClass | aClass subclasses];
                    roots: roots)
```

Analysis

In the basic steps of this example we have added a TreeView widget to a canvas and configured it to adapt to the inheritance hierarchy of the classes present in your image. The key point of this was to provide and initialize an instance of *TreeAdaptor*. This TreeAdaptor instance is initialized with a *childrenBlock* which returns the subordinate entries to a given node. This block is being evaluated whenever a node is expanded for the first time. It returns a collection of sub entries, the children of the expanded nodes. Each entry in this collection is then being wrapped by an instance of *TreeNode*. This TreeNode instance is manufactured automatically in the TreeAdaptor.

The collection of resulting TreeNodes are kept in the *children* instance variable of the parent's TreeNode. When a node which has previously been expanded will be expanded again (after being collapsed), its collection of children will be obtained from that instance variable. The children block won't be evaluated a second time. However, situations may occur in which a re-evaluation of the children block for already fetched children becomes necessary. You can achieve this by sending the TreeAdaptor an invalidate message such as in:

```
    classes list invalidate: Collection.
```

Expanding and collapsing entries programmatically

Strategy

Applications employing TreeViews will often have to access entries within the tree view in order to expand or collapse them as a reaction to some user interaction external to the tree view widget. In this example we will add a list menu to the tree widget containing menu commands for expanding and collapsing selected entries.

Tutorial Steps

Tutorial Example: *SimpleClassBrowser*

1. In the Properties Tool's Basics Page, fill in the TreeView's **Menu** property with the name of the method that will return an appropriate menu. (In this example we'll call it *classesMenu*)

2. Use the Menu Editor to create and install the *classesMenu* with the following entries:

Label	**Value**
Expand	*expand*
Collapse	*collapse*
Expand / Collapse Subtree	*toggleExpandSubtree*
Expand All	*expandAll*

Basic Steps

Tutorial Example: *SimpleClassBrowser*

1. Use a System Browser to create the methods for the menu commands given above:

```
expand
    classes list expandAt: classes selctionIndex
```

```
collapse
    classes list collapseAt: classes selectionIndex
```

```
toggleExpandSubtree
    classes list toggleExpandSubtreeAt: anIndex
```

```
expandAll
    classes list expandAll
```

Variant: Using *SelectionInTree* instead of *SelectionInList*

In order to provide a more convenient way to access, expand and collapse the selected entry, a subclass of *SelectionInList* is provided, called *SelectionInTree*. This class enhances SelectionInList by some methods to give you easier access to selected entries and nodes.

1. Replace *SelectionInList* in the initialize method by *SelectionInTree* as shown below:

```
initialize
    super initialize.
    classes := SelectionInTree with:
        (TreeAdaptor new
            childrenBlock: [:aClass | aClass subclasses];
            root: Object)

    classes list expand:list rootNode
```

2. Now you can change some of the expand/collapse methods as follows:

```
expand
    classes expandSelectedNode.
```

```
collapse
    classes collapseSelectedNode
```

Analysis

You can use the messages in the TreeAdaptor class's 'expand-collapse' protocol to expand and collapse entries programmatically. Note that these messages all send *changed:* to the TreeAdaptor which cause a corresponding *update*: message to be sent to the TreeView. This allows the TreeView's display to update appropriately whenever an entry is being expanded or collapsed.

Expanding along a path

Strategy

Entries in a TreeView can only be expanded if they have been retrieved before by a children block. Sometimes however, an application needs to expand up to a certain object which has not yet been retrieved. Often, not even the parent objects of the requested object have been retrieved and expanded.

Thus, expanding and selecting up to the requested objects needs all parent objects to be expanded. Class TreeAdaptor supports this through message *expandPath:*, which expects a collection of objects comprising the components of the path.

Tutorial Steps

Tutorial Example: *SimpleClassBrowser*

1. In our example we will enhance the *classesMenu* created before by a new menu item to search for a class. Use the Menu Editor to do this as follows:

Label	**Value**
Find Class…	*findClass*

Basic Steps

Tutorial Example: *SimpleClassBrowser*

1. Use a System Browser to create the notification method *findClass* as shown below:

```
findClass

    | aClass |
    aClass := Smalltalk at:className value ifAbsent: [^self].
    Classes list expandPath: aClass withAllSuperclasses reverse.
    Classes selction: aClass.
```

Analysis

In order to use *expandPath*, your application must be able to determine all the parents to a certain entry up to the root of the hierarchy.

Adding and removing entries in a hierarchy

Strategy

Applications employing TreeViews will sometimes have to add and remove entries. Adding means that a new entry is to be inserted somewhere in the hierarchy. This implies that it has to be specified where the entry should be added, more precisely, which the entry's parent should be. Removing means that all subordinate entries — if any — of a selected entry will be removed, too.

Tutorial Steps

Tutorial Example: *SimpleClassBrowser*

1. In our example we will enhance the *classesMenu* created before by two menu items Add and Remove. Use the Menu Editor to do this as follows:

Label	**Value**
Add…	*AddEntry*
Remove	*RemoveEntry*

Basic Steps

Tutorial Example: *SimpleClassBrowser*

1. Use a System Browser to create the methods for the new menu commands given above:

```
addEntry

    | newEntry selectedEntry |
    selectedEntry := classes selction.
    newEntry := Dialog request: 'Enter a name for the new sub
    entry'.
    newEntry isEmpty if True: [^self].
    newEntry := Smalltalk at: newEntry ifAbsent: [newEntry].
    classes list
        add: newEntry asChildOf: selectedEntry;
        expand: selectedEntry.
   classes selection: newEntry.
```

```
removeEntry

    classes list removeAtIndex: classes selectionIndex
    "Or:  classes list remove: classes selection."
```

The code for *addEntry* not only adds a new entry, but also selects it by first making sure that the parent entry is expanded, and then updating the selection holder. Note also, due to dependency mechanisms supported by TreeAdaptor's add and remove messages, the widgets display updates appropriately whenever entries are inserted or removed.

Variant

The *addEntry* method given above simply adds class or string objects to the hierarchy of classes displayed in the TreeView widget. If you try to expand one of these new entries, you will run into an error. This is because the TreeAdaptor's children block expects all entries to be of instances of *Class*. As a workaround we may prevent children block from being evaluated for those String entries by explicitly adding information that these entries do not have children. We do so by explicitly constructing and adding an instance of *TreeNode* instead of letting TreeAdaptor providing for a default TreeNode to wrap and manage the new entry:

```
addEntry

    | newEntry selectedEntry newNode |
    selectedEntry := classes selction
    newEntry := Dialog request: 'Enter a name for the new sub entry'.
    newEntry isEmpty if True: [^self].
    newEntry := Smalltalk at: newEntry ifAbsent: [newEntry].
    newNode := newEntry asTreeNode hasChildren: false.
    classes list
        add: newEntry asChildOf: selectedEntry;
        expand: selectedEntry.
    classes selection: newEntry.
```

This ensures that the new tree node is marked as having no children. However, you can still add new entries to those marked as having no children earlier. When you do so, you consequently can also expand and collapse the String entries added manually.

Pre-fetching child nodes

Strategy

Sometimes, you won't want the children of a certain node to be fetched dynamically through evaluation of a children block. Instead you may want to provide the children in advance and to statically assign them to the node. In our example this is the case for the direct children of the root entry. The root entry will be some kind of dummy labeled 'Volumes'. Its children are the volumes in your file system. Since we know this and want the volumes to be displayed at startup, we can as well fetch them in advance. At last, this also keeps our children block simpler. All other children being fetched through the children block are then entries in a directory.

Basic Steps

Tutorial Example: *SimpleFileBrowser*

1. Use a System Browser to edit an *initialize* method which provides and initializes the TreeAdaptor instance used as the tree view widget's aspect:

```
initialize

    | treeAdaptor root |
    root := TreeNode for: #root label: 'Volumes'.
    root children: (Filename volumes
        collect: [:each | TreeNode for: each asFilename label: each]).

    treeAdaptor := TreeAdaptor new
        childrenBlock: [:aDirectory | self childrenOf: a Directory];
        root: root;
        expandAt: 1

    tree := SelectionInList with: treeAdaptor
```

Analysis

You can use method *children*: to assign a collection of children to a node statically. This collection will be kept in the tree node's instance variable *children*. When this node is to be expanded, TreeAdaptor detects this and does not evaluate the children block. Thus, in you children block, you do not have to care about those entries which have statically assigned children. This keeps your children block leaner.

Variant

Instead of constructing the collection of child entries explicitly and assign it to a node. You can also have the TreeAdaptor do this based on the children block provided on initialization. This is an example:

```
    aTreeAdaptor fetchChildrenFor: aTreeNode
```

Explicit manufacturing of tree nodes

Strategy

In the previous class browser example we have shown how to initialize a tree view widget with a TreeAdaptor which itself was provided with a children block to adapt to the application specific hierarchy. Therefore we initialized the TreeAdaptor with a children block that returned a collection of child entries to a parent entry to be expanded.

A TreeAdaptor wraps all entries returned by the children block in instances of TreeNode. The real entry wrapped by a TreeNode is called a TreeNode's "subject". Basically, a TreeNode keeps track of an entry's expand/collapse status. It also provides various options to store additional information about an entry. These options allow to assign explicitly:

- the type of node (is used to determine the icon to display)
- a specific icon for this node (overrides icon detection via type)
- a specific display string to use as the entry's label
- information about whether children can be fetched or not

When a TreeAdaptor automatically manufactures TreeNodes for entries fetched through a children block, these TreeNodes are initialized with default values which are:

- type of node: *#folder*
- display string: the entry's *displayString*
- icon: determined according to the node's type
- has children: *true*

In order to explicitly set one of these options, we may initialize TreeAdaptor with a children block that already wraps the child entries with TreeNodes.

Basic Steps

Tutorial Example: *SimpleFileBrowser*

2. Use a System Browser to edit an *initialize* method which provides and initializes the TreeAdaptor instance used as the tree view widget's aspect:

```
initialize

    | treeAdaptor root |
    root := TreeNode for: #root label: 'Volumes'.
    root children: (Filename volumes
            collect: [:each | TreeNode for: each asFilename label: each]).

    treeAdaptor := TreeAdaptor new
            childrenBlock [:aDirectory | self childrenOf: a Directory];
            root: root;
            expandAt: 1
    tree := SelectionInList with: treeAdaptor
```

The children node block[1] provided calls method *childrenOf:* which is implemented as shown below. When this block is evaluated as a reaction to an expand action initiated by the user, the block parameter *each* is the Filename instance comprising the parent node to fetch the children for.

```
childrenOf: a Directory

    ^aDirectory directoryContents
        collect:
        [:each || file |
        file := aDirectory construct: each.
        (file asTreeNode) label: each; isParent: self isDirectory: file)]
```

The method returns a collection of instances of *Filename*, each wrapped in a specific TreeNode. Fetching the children is primarily done through Filename's *directoryContents* method which returns a collection of strings. These strings are the local file names relative to the parent directory. They are expanded to full path names (which is necessary, because they will be used in successive expands) using the *construct:* method.

Finally an instance of TreeNode is created for each resulting child Filename. The TreeNode's label (its display string) is set to the local file name, omitting the leading path name. Furthermore, method *isParent:* is used to tell the TreeNode whether children can be fetched or not. We do the latter to suppress evaluation of the children block for entries which are not directories.

[1] We will use the term "children node block" to refer to a block returning a collection of TreeNode instances, each wrapping one of the children of an entry to be expanded. On contrary we will use the term "children block" to refer to a block which returns the children objects themselves (not yet wrapped in TreeNodes).

Analysis

There are two reasons in this example, which make it necessary to use a children node block instead of a children block. One reason is that we need to supply the TreeNodes with alternative label strings (different from what the entries' *displayString* would provide). If we wouldn't do this, the entries' labels would be full path names (This is what is returned by a Filename's *displayString* method.) The other reason is that we have to use *isParent:* to pre-determine that children shall only be fetched for directories, not for files. We wouldn't be able to do this with a simple children block. Note that in the example above, message *isParent: false* has the side effect of setting a TreeNodes *type* to *#leaf* instead of the default value of *#folder*. This in turn causes the pre-installed leaf icon to be displayed for those entries, which is desired in the case of this example. However, if you only want to specify that a TreeNode definitely has no children, without impacting the type of node, you may use message *hasChildren:* instead of *is-Parent:*.

Using identity comparison for TreeNodes

Strategy

When a TreeNode's subject has to be looked up in a TreeAdaptor's collection of nodes, this is done by the means of equality comparison. I.e. if you send *indexOf: anObject* to a TreeAdaptor, each node's subject is checked whether it's **equal** to *anObject*. Alternatively, you can provide for identity comparison to be used by wrapping the entries in instances of *IdentityTreeNode* instead of *TreeNode.*

Basic Steps

Tutorial Example: *SimpleClassBrowser*

1. In method *initialize* provide a TreeAdaptor with a children block that returns instances of *IdentityTreeNode*.

```
initialize

    super initialize.
    classes := SelectionInList with:
        (TreeAdaptor new
            childrenBlock [:aClass | aClass subclasses
                collect: [:each | each asIdentityTreeNode]];
            root: Object asIdentityTreeNode).
    classes list expand: classes list rootNode.
```

Variant

In the method above message *asIdentityTreeNode* is used to manufacture the instances of IdentityTreeNode for the root entry and for each child entry. However, it is sufficient to only provide the root entry as a TreeNode and let the TreeAdaptor manufacture all the children nodes accordingly: Each time a new TreeNode has to be created, TreeAdaptor will use the root entry's TreeNode class. Thus the following would be sufficient:

```
initialize

    classes := SelectionInList with:
        (TreeAdaptor new
            childrenBlock [:aClass | aClass subclasses];
            root: Object asIdentityTreeNode).
    classes list expand: classes list rootNode.
```

Using a multi-parent tree adaptor

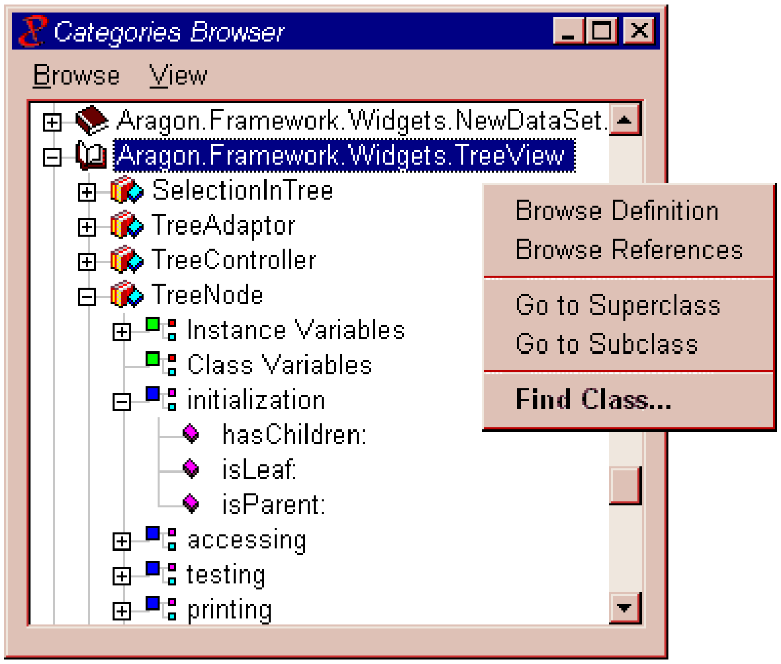
Strategy

To adapt to a multiple parent hierarchy you would use a *MultipleParentTreeAdaptor* instead of its superclass *TreeAdaptor* in your ApplicationModel.

Basic Steps

1. ... PENDING ...

Tutorial Example 2: A Class Protocol Browser



We will now build a more comprehensive example dealing with some more facilities provided by tree view widgets. Subject of this example will be a simple combined class categories and protocols browser. The application will use a single tree view to display a hierarchy, the top level of which are the class categories in your VisualWorks image. The second level are the classes within these categories. The third level are the method protocols within each class, as well as one folder for each, the instance and class variables. The fourth and final level are the methods with each protocol of a class or the instance and class variables respectively. The user can browse through the classes, their instance variables, class variables and methods by expanding first a class category, then a class entry, a protocol, etc.

The particularity of this example application is, that there are entries of different kinds placed in one single hierarchy, displayed in one single tree view. This raises the necessity to find a way to distinguish the different kinds of entries. This will be done by using different icons for each type of entry.

Using a two parameters children block

Strategy

This example includes a rather complex children block: In each fetch for children we will have to first detect the type of parent entry for which children are to be fetched, and then do the appropriate action. Therefore, each entry must be equipped with a tag to distinguish the type of object. We will use TreeNode's instance variable *type* for this purpose.

The block given as parameter to *childrenBlock* can either have one block parameter or two. In case of a two parameter block, the second parameter will be set to the type of the node/entry to be expanded.

Basic Steps

Tutorial Example: *CategoriesBrowser*

2. In method *initialize* the tree view's TreeAdaptor is initialized with a children node block that takes two parameters, which are the nodes's subject (the entry itself) and its type. This block simply sends message *childrenFor:type:* in order to fetch for children of the entry to be expanded. The hierarchy's root is a single dummy entry #*root* with label „Class Categories".

```
initialize

    super initialize
    categories := SelectionInTree with:
        (TreeAdaptor new
            childrenBlock:
                [:anEntry :type | self childrenFor: anEntry type: type]).
            root: (TreeNode for: #root label: 'Class Categories');
            expandAt: 1)
```

Using user defined icons

Strategy

Each entry in a tree view can be displayed with an associated icon. Moreover, a different icon can be displayed when an entry is expanded. Generally, which icon will be displayed is determined by the nodes type. Each TreeView can therefore be equipped with an image list, a dictionary of icons with the types of nodes as the dictionary keys. If no image list is specified explicitly, TreeView uses a default image list. This default image list has entries for the pre-defined types *#folder* and *#leaf*. It is configured in *TreeView class>> initialize* and can be accessed through *TreeView class>> defaultImageList*. This is how the default image list is defined:

```
initialize
    "Initialize the default image list"

      DefaultImageList := IdentityDictionary new
              at: #folder    put: self folderIcon -> self open folderIcon;
              at: #leaf       put: self leafIcon;
      yourself.
```

Basic Steps

Tutorial Example: *Categories Browser*

3. In the application model's postBuildWith: method configure the TreeView instance with a user defined image list:

```
postBuildWith: aBuilder

    | myImageList |
    myImageList := IdentityDictionary new
           add: #category  ->  (TreeView defaultImageList at: #folder);
           add: #class       ->  (self class classIcon ->self class openClassI-
           con);
           add: #variables ->  self class variablesIcon;
           add: #protocol   ->  self class protocolIcon;
           add: #method    ->  self class methodIcon;
           add: #instvar     ->  self class instvarIcon;
           add: #classvar   ->  self class instvarIcon;
    yourself.
    (builder componentAt: #categories) widget imageList: myIma-
    geList.
```

4. In the children node block ensure that each TreeNode is initialized with a correct type, selected from one of the keys in your image list. In our example the children node block dispatches the fetch for children to different methods specialized for fetching children of certain types. This is how the fetch for children of a class entry looks like:

```
childrenForClass: aClass

    | answer |
    answer := .OrderedCollection new .

    answer add: ((TreeNode for: (#instvars -> aClass)) type: #variables;
            displayString: 'Instance Variables').

    answer add: ((TreeNode for: (#classvars -> aClass)) type: #variables;
            displayString: 'Instance Variables').

    answer addAll: (aClass organization categories
            collect:
                [:each | (TreeNode for: (each -> aClass))
                type: #protocol;
                displayString: each]).

    aClass isMeta ifFalse: [answer add: ((TreeNode for: aClass class)
                type: #class)].
    ^answer
```

Analysis

An image list is a dictionary with the node types as the dictionary keys. The entries' values are either a single image or an Association of two images. If there is only one image specified for a type, this image will be used in both collapsed and expanded state of an entry. If there is an association of two images given for a type, the first one (key) will be used for entries in collapsed state and the second one (value) for entries in expanded state.

TreeView provides a default image list with entries for the pre-defined types *#folder* and *#leaf*. Type *#folder* is the default type of each TreeNode as being initialized by TreeNode's instance creation methods (*for:label*). Type *#leaf* will be used if you send a TreeNode the message *isLeaf: true* (or *isParent: false*, resp.). You can access the pre-built icons in class TreeView with the messages *folderIcon, openFolderIcon* and *leafIcon*.

Selecting an entry's parent entry

Strategy

In applications employing a tree view with large hierarchies it may be useful to provide the user with a facility to jump from the currently selected entry in a tree view to that entry's parent entry. In the categories browser example we do this by a menu entry "Go to -> Parent"[2].

Basic Steps

Tutorial Example: *CategoriesBrowser*

1. Get the currently selected node from the *SelectionInTree* instance using method *selectedNode*.
2. Get the node's parent node using *aNode parent*.
3. Change selection to the parent node's subject.

```
gotoParent

    | selectedNode parentNode |
    selectedNode := categories selectedNode.                  "Step 1"
    parentNode := selectedNode parent.                        "Step 2"
    categories selection: parentNode subject.                 "Step 3"
```

Updating a tree view's contents

Strategy

When a hierarchy displayed in a tree view in result to some external actions you will have to update the tree view’s display. This means re-fetching children of nodes which have previously been expanded. Since a tree node caches the children of its entry, you will need to tell it to drop the cached

[2] Since the TreeView already comes with built-in support for moving the selection to the currently selected entry's parent entry(keyboard shortcut C*trl Left*), there is no real need for this function. But anyway,...

children and re-fetch them using the children block. You can use the TreeAdaptor's *invalidate* messages to do so.

Basic Steps

Tutorial Example: *CategoriesBrowser*

1. Send message invalidate to the TreeAdaptor

```
refreshDisplay

    categories list invalidate.
    categories list expandAt: 1.
```

Variant A: Invalidating and re-expanding the currently expanded entries

A problem with the solution shown in the basic steps is that all the expanded entries and branches will collapse and your current selection gets lost. You can choose to re-expand all the entries which were expanded before the invalidation. TreeAdaptor supports this by method *invalidate-AndReExpand*. This method will first collect the subjects of all the currently expanded entries. Then an invalidation is performed. After that the new contents are searched for the entries which were expanded before. Each entry found will be expanded again. Additionally, this keeps the selection in the right place. If a formerly expanded entry can't be found anymore in the place it was before, it is ignored.

```
refreshDisplay

    parcels list invalidateAndReExpand.
```

Variant B: Invalidating only a part of an hierarchy

Sometimes you may not want to invalidate the whole hierarchy, but rather a part of it, of which you know something has changed. You can do this by invalidating only the subtree beyond a certain entry or node. TreeAdaptor provides three messages, *invalidate: anEntryOrNode* and *invalidateAt: anIndex* for this purpose. Here is an example.

```
invalidate: aFile

    "Something of aFile has changed. Ensure that the children block is re-
    evaluated for the file's directory entry in order to retrieve the correct
    information about aFile.

    | parent |
    parent := aFile directory.
    hierarchy list invalidateAndReExpand: parent.
```

You can also use a variant of invalidate to re-expand the currently expanded entries after an invalidation:

```
invalidate: aFile

    "Something of aFile has changed. Ensure that the children block is re-
    evaluated for the file's directory entry in order to retrieve the correct
    information about aFile.

    | parent |
    parent := aFile directory.
    hierarchy list invalidateAndReExpand: parent.
```

Customizing the overall behavior

Strategy

Several overall settings can be customized with class methods. In particular you can customize ...

- selection hiliting
- handling of re-selecting an entry
- when to display the opened folder symbol

Customizing Selection Hiliting

Display of selection hiliting can be customised selectively in class *EnhancedSequenceView* and its subclasses (*TreeView*, *NewDataSetView*) by sending *useStandardHiliting: aBoolean* to the respective class. The default is to display the selected entry/entries in a reverse appearance regardless of whether the widget has the input focus or not (*useStandardHiliting: true*). This is also the default behaviour of list boxes in VisualWorks.

You can configure class *EnhancedSequenceView*, and hence all its subclasses — namely *TreeView* and *NewDataSetView* —, or each subclass selectively to adopt the Win95-style behaviour of hiliting the selected entry only if the respective widget has the input focus by sending *useStandardHiliting: false*.

Customizing handling of re-selecting an entry

Handling of repeated selection can be customised selectively in class *EnhancedSequenceView* and its subclasses (*TreeView, NewDataSetView*) by sending *deselectOnReselection: aBoolean* to the respective class. The default is to deselect a selected entry if it is re-selected, with the subclasses sharing this configuration with *EnhancedSequenceView*.

Customizing when to display the opened folder symbol

As a default, the opened folder symbol (or user-defined *opened* symbols) are displayed only for the selected entry. This is in line with the behaviour of Tree Controls in Windows 95. You can change this behaviour to display the opened folder symbol for each expanded entry by evaluating:

```
TreeView displayOpenFolderWhenSelectedOnly: false.
```

Object Reference

This section is a reference manual giving detailed description of the *TreeView Widget* programming interface. Only the *public* classes, protocols and methods are listed and explained, comprising the programming interface for deployment and reuse. All the classes and methods not listed in here should be considered private to the implementation and are subject to change in future releases.

Overview

The classes providing the functionality of TreeView widgets are mainly a View/Controller pair along with special adapter classes providing the means to express and adapt to varying hierarchical relationships in a uniform way. In particular those classes are:

Internal Classes

TreeView

is responsible for handling the view part of the widget,
inherits from *SequenceView*.

TreeController

is responsible for handling the user interaction,
inherits from *SequenceController*.

TreeViewSpec

makes the new widget accessible from Palette Tool and Canvas Painter,
inherits from *SequenceViewSpec*.

TreeViewDirectorySelector

implements a TreeView-based directory selector dialog,
inherits from *SimpleDialog*.

Adaptor Classes

SelectionInTree

Instances may serve as specialised aspect models in an ApplicationModel,
inherits from *SelectionInList.*

TreeAdaptor

provides the means to express and adapt to arbitrary and varying hierarchical relationships in a uniform way,

inherits from *SequencableCollection.*

TreeNode

Instances wrap domain model objects to be displayed in a TreeView. Uses equality comparisons on node subjects.

IdentityTreeNode

A subclass of *TreeNode* that allows to adapt to hierarchies based on object identity comparisons rather than object equality.

MultipleParentTreeAdaptor

A subclass of *TreeAdaptor* that provides additional operations for dealing with non-strict hierarchies, i.e. hierarchies containing entries that may have more than one parent entry.

In the very most cases you won't be concerned much about the internal classes. Instead you will employ a tree view widget by adding it from the Palette Tool to a user interface canvas and configuring it in Properties Tool.

In this chapter we will describe the public interface of each class containing the messages, which may be used in an application employing a tree view. Messages not described here are considered private and subject to change.

How to read the methods

The descriptions of the various methods are given in a certain schema. They consist of a title with the method selector(s) and parameters, an explanatory text and one or more closing lines denoting possible exceptions that can be raised during execution of this method and the method's return value. An example:

indexOf: *anElement*
indexOf: *anElement* ***ifAbsent:*** *notFoundBlock*

Searches for anElement among the entries in the receiver's hierarchy (the nodes' subjects). **Equality comparison (=)** is used to compare the entries. Only entries in expanded branches are searched. If no matching entry can be found, either signal *notFoundError* is raised (variant 1) or the notFoundBlock is evaluated.

! *notFoundError* when no matching entry was found
(only in variant one).
^ *<TreeNode>* | ***nil***

You can read this method description as follows:

Described are two separate but similar methods with identical purpose but a varying number parameters. Usually one method calls the other one with some default values for missing parameters.

During execution an exception with Signal *notFoundError* might be raised. You should somehow handle those exceptions in your code.

The method's return value is guaranteed to be either an instance of class *TreeNode* or *nil*. If there is no closing line specifying the return value of a method. This method is supposed to return *self*.

Class SelectionInTree

Inherits from: *SelectionInList*

This class enhances *SelectionInList* by some methods to give you more convenient access to selected entries and nodes. SelectionInTree instances are initialized with an instance of TreeAdaptor, such as in *SelectionInTree with: aTreeAdaptor*.

Accessing

selectedNode

Get the currently selected node.
^ *list nodeAt: selectionIndex* *<nil or TreeNode>*

Expand - Collapse

expandSelectedNode

Expand the currently selected node. Implemented as *list expandAt: selectionIndex*

collapseSelectedNode

Collapse the currently selected node. Implemented as *list collapseAt: selectionIndex*

toggleExpandSelectedNode

Toggle the expand status of the currently selected node. Implemented as *list toggleExpandAt: selectionIndex*

Selection in List

selectParent

Move selection to the parent node of the currently selected entry. The returned value is the subject of the selected parent node or nil if no parent node was found.
^ *<Object>* **|** ***nil***

Class TreeAdaptor

Inherits from: *SequenceableCollection*

TreeAdaptors provide the means to express and adapt to varying hierarchical relationships in a uniform way. In principal a TreeAdaptor is configured with an object comprising the root(s) of the hierarchy and a block — the so called children block — that is responsible for retrieving and returning a collection of subordinate hierarchy entries or nodes (the children) to a given node (the parent). TreeAdaptors are used as the subject (list) to a *SelectionInList* or *SelectionInTree* for a tree view.

TreeAdaptor inherits from *SequencableCollection* and thus provides you with the familiar enumeration and accessing protocols used with collections. Not least, this makes it possible for instances of TreeAdaptor to be used as subjects to a SelectionInList.

aNodeOrAnObject

Many of the messages described below either expect an Instance of TreeNode or a domain model entity (a node's subject) as a parameter. In case of the latter, the TreeAdaptor has to first look up the corresponding TreeNode. This is done by comparing each TreeNode's subject to the provided element using the object equality operator message (=)[3].

On the contrary if the parameter specifies an instance of TreeNode, the node is uniquely identified, and no search for the TreeNode is required. Thus, it is strongly recommended that you provide instances of TreeNode

[3] Note that the object identity operator (=) is overloaded in *TreeNode* to compare the node subjects. Furthermore, it is overloaded in *IdentityTreeNode* to use object identity comparison on the subjects instead of object equality.

instead of plain elements as parameters for the respective messages wherever possible.

Variables

roots *<Collection>*

The hierarchy's roots. Can be accessed via *roots* and *roots:*. A tree view can display a hierarchy with more than on root entries. However, most often, there will only be one root entry corresponding to *roots* referring to a one-element collection. Therefore convenience methods are provided (*root, root:, rootNode*), which assume that there is only one element in *roots*. The first root's species determines the TreeNode class to use for automatic manufacturing of tree nodes.

childrenBlock *<BlockClosure>*

This holds a block which is responsible for fetching the children of a given entry and wrapping each children in an instance of *TreeNode*. The block can take one parameter, the subject of the node to fetch children for, or two, wherein the second parameter is set to the parent node's type. The block returns a collection of child entries. The elements of this collection may either be plain entries, in which case TreeNodes will automatically be created for each entry, or TreeNodes, each of which wrapping a child entry. This collection may be empty if no children are found.

Initialise – Release

childrenBlock: *aBlockClosure*

Set the *childrenBlock* to be aBlockClosure. The block can take one parameter — the subject of the node to fetch children for — or two, wherein the second parameter is set to the parent node's type. The block's return value is expected to be a collection of child entries which may be empty. The elements may already be wrapped in instances of *TreeNode*. If not, a TreeNode will automatically be created for each entry. In this case, the automatically manufactured TreeNode will be of the same class (*species*) as the first root entry.

roots: *aCollection*

Set the *roots* of the hierarchy to be aCollection. The elements in the hierarchy can be instances of TreeNodes or any other kind of Object (plain entries). If the elements are not TreeNodes they are automatically wrapped with TreeNodes.

root: *anNodeOrAnObject*

Convenience for: ***roots:*** *(Array with: anNodeOrAnObject)*

Accessing

childrenBlock

Get the receiver’s children block.
^ ***childrenBlock*** *<BlockClosure>*

fetchChildrenFor: *aNode*

Uses the configured children block to fetch the children for *aNode*. This method is generally used internally in a tree adaptor. Applications may use this method in order to provide for pre-fetching child entries.
^*the resulting collection of child nodes <Collection>*

at: *anIndex*

Get the hierarchy entry at the given index. The index corresponds to the index of the entry as displayed in the tree view. Entries are indexed depth-first starting at the first root. Only entries in expanded branches are indexed. The returned value is the subject of the TreeNode at anIndex.
^*the node's subject <Object>*

indexOf: *aNodeOrAnObject*
indexOf: *aNodeOrAnObject* ***ifAbsent:*** *notFoundBlock*

Searches for *aNodeOrAnObject* among the entries in the receiver’s hierarchy and returns its index. **Equality comparison (=)** is used to compare the entries if *aNodeOrAnObject* is a node's subject, **identity comparison (==)** is used if it is an instance of *TreeNode*. Only entries in expanded branches are searched. If no matching entry can be found, either signal *notFoundError* is raised (variant 1) or the notFoundBlock is evaluated.
! ***notFoundError*** when no matching entry was found
(only in variant one).
^ *<TreeNode>*

nodeAt: *anIndex*

Get the node at the given index. This method does quite the same as *at: anIndex* does, only in this case it returns not the node’s subject, but the node itself. Note: Only entries in expanded branches are in-

dexed.
^ *<TreeNode>*

nodeFor: *anObject*
nodeFor: *anObject* ***ifNone:*** *notFoundBlock*

Searches for anObject among the entries in the receiver's hierarchy (the nodes' subjects). **Equality comparison (=)** is used to compare the entries. All the already fetched entries in both, collapsed and expanded state, are compared. If no matching entry can be found, either signal *notFoundError* is raised (variant 1) or the notFoundBlock is evaluated.
! ***notFoundError*** when no matching entry was found (only in variant one).
^ *<TreeNode>*

roots

Get the collection of root entries.
^ ***root*** *<Collection<TreeNode>>*

root

Get the first root entry. This is a convenient method for hierarchies which do have only one root. The returned value is the first root node's subject.
^ ***root*** *<Object>*

rootNode

Get the first root node. This is a convenient method for hierarchies which do have only one root.
^ ***root*** *<TreeNode>*

Enumerating

The messages in this protocol allow to enumerate over the elements or nodes within the receiver's hierarchy. In each case, only expanded branches of the hierarchy are visited. Based on enumeration method **do:**, all the other enumeration messages inherited from SequenceableCollection (collect:, select:, reject:, etc.) are available for instances of TreeAdaptor too.

countChildrenAt: *anIndex*
countChildrenOf: *aNodeOrAnObject*

Count all the children and children's children in the subtree starting at *anIndex* or at *aNodeOrAnObject*. Only expanded branches are

visited.
^ *<Integer>*

detect: *aBlock*
detect: *aBlock* ***ifNone:*** *anotherBlock*

Evaluate aBlock with each of the receiver's elements (node subjects) as the argument. Answer the first element (node subject) for which aBlock evaluates to true. The block may optionally expect two arguments, in which case each element's indent level within the hierarchy is passed as the second block parameter.

The variants allow to specify a block to evaluate when no matching element is found. Only expanded branches are visited.
! *notFoundError* when no entry was found (only in variant 1 and 3).
^ *<Object>*

detectNode: *aBlock*
detectNode: *aBlock* ***ifNone:*** *anotherBlock*
detectNode: *aBlock* ***startWith:*** *aNode*
detectNode: *aBlock* ***startWith:*** *aNode* ***ifNone:*** *anotherBlock*

Evaluate aBlock with each of the receiver's nodes as the argument. Answer the first node for which aBlock evaluates to true. The block may optionally expect two arguments, in which case each node's indent level within the hierarchy is passed as the second block parameter.

The variants allow to specify a block to evaluate when no matching entry is found and/or the node to start searching at. If a start node is given only the subtree beyond this node is searched. Only expanded branches are visited.
! *notFoundError* when no entry was found (only in variant 1 and 3).
^ *<TreeNode>*

do: *aBlock*
do: *aBlock* ***startWith:*** *aNode*
do: *aBlock* ***startWith:*** *aNode* ***indent:*** *anotherBlock*

nodesDo: *aBlock*
nodesDo: *aBlock* ***startWith:*** *aNode*
nodesDo: *aBlock* ***startWith:*** *aNode* ***indent:*** *anotherBlock*

Evaluate aBlock with each of the receiver's elements (node subjects for variants 1 to 3 and nodes for variants 4 to 6) as the argument. The block of variants 1,3,4 and 6 may optionally expect two arguments, in which case each elements indent level within the hierarchy is passed as the second block parameter.

The variants allow to specify a node to start evaluation at and additionally an initial indent value. If a start node is given only the subtree beyond this node is visited. Only expanded branches are visited.

Expand – Collapse

The messages in this protocol allow to expand or collapse entries in a hierarchy. Notifications are send to the TreeView(s) displaying the receiver's hierarchy to automatically update the tree view's display.

expand: *aNodeOrAnObject*
collapse: *aNodeOrAnObject*
expandAt: *anIndex*
collapseAt: *anIndex*

Expand or collapse a specified entry. Variants 1 and 2 expect an instance of TreeNode or an element (a node's subject) as parameter. Variants 3 and 4 use *nodeAt:* to search for the element to expand or collapse. Note: Only entries in expanded branches are indexed.

expandSubtree: *aNodeOrAnObject*
collapseSubtree: *aNodeOrAnObject*
expandSubtreeAt: *anIndex*
collapseSubtreeAt: *anIndex*

Expand or collapse all the entries in the subtree starting at the specified entry. Variants 1 and 2 expect an instance of TreeNode or an element (a node's subject) as parameter. Variants 3 and 4 use *nodeAt:* to search for the element to expand or collapse. Note: Only entries in expanded branches are indexed.

expandAll
collapseAll

Expand or collapse all the entries in the whole hierarchy.

toggleExpand: *aNodeOrAnObject*
toggleExpandAt: *anIndex*
toggleExpandSubtree: *aNodeOrAnObject*
toggleExpandSubtreeAt: *anIndex*

Toggles the expand status of a single hierarchy entry or all the entries in a subtree. See expand/collapse messages for further explanations.

expandPath: *aCollectionOfNodesOrElements*

Expand the entries along a path. The path is specified by the entries or nodes in the parameters. The first element/node in this collection is expected to be already fetched, so that there is already a node for it contained in the receiver's hierarchy. In case of variant 2 however, all nodes in this collection are expected to be already fetched and present. Expanding will happen from the first to the last collection element.

expandedEntries
expandedNodes
expandedEntriesStartingAt: *aNode*
expandedNodesStartingAt: *aNode*

Return a collection of all the expanded entries (Variants 1 and 3) or nodes (Variants 2 and 4) in the hierarchies. Variants 3 and 4 allow to specify a starting node. These methods are most commonly used for invalidation tasks after which the formerly expanded entries would have to be expanded again.

Invalidating

invalidate
invalidate: *aNodeOrAnObject*
invalidateAll: *aCollectionOfNodesOrObjects*
invalidateAt: *anIndex*

Invalidate certain entries. This means that the children cache is released and a re-evaluation of the children block for the nodes is forced to happen the next time these nodes are expanded. Variant 1 invalidates all root nodes. Variants 2 and 4 invalidate a certain node. Variant 3 invalidates all the nodes in a collection.
! *notFoundError* when no entry was found (only in variant 1 and 3).

invalidateAndReExpand
invalidateAndReExpand: *aNodeOrAnObject*
invalidateAndReExpandAll: *aCollectionOfNodesOrObjects*
invalidateAndReExpandAt: *anIndex*

Provides a convenient way to update a tree view's contents while keeping the currently expanded entries expanded after the invalidation, provided the expanded entries are still present in the hierarchy visualised by that tree view. Variant 2, 3 and 4 invalidate only part of an hierarchy starting at the denoted entry/entries (an index or a TreeNode or a subject), whereas Variant 1 invalidates the whole hierarchy.

Modifying

The messages in this protocol allow to add and remove entries to a hierarchy. Notifications are send to the TreeView(s) displaying the receiver's hierarchy to automatically update the tree view's display.

add: *aNodeOrAnObject* ***asChildOf:*** *aParentNodeOrObject*
add: *aNodeOrAnObject* ***asChildAt:*** *anIndex*

Adds a new entry to the hierarchy to the children of a certain entry. The new entry may be an instance of TreeNode or a plain entry, in which case the new entry will be wrapped by an automatically created TreeNode. Variant 1 allows to specify the parent element or node. Variant 2 specifies the parent node by index (See message *at:* for descriptions how indexes are interpreted). If the children of the parent node have not yet been fetched, this is done before the new entry is added.
! *notFoundError* when the parent entry was not found (only in variant 1 when the first parameter is not a tree node).

remove: *aNodeOrAnObject*
remove: *aNodeOrAnObject* ***ifAbsent:*** *notFoundBlock*
removeAtIndex: *anIndex*

Remove an entry from the hierarchy. The new entry may be referred to either by an instance of TreeNode, by an element or by index (See message *at:* for descriptions how indexes are interpreted). Variant 2 allows you to specify a block to evaluate, when the entry to delete can not be found.
! *notFoundError* when the parent entry was not found (only in variant 1 when the first parameter is not a tree node).

Class TreeNode

Inherits from: *Object*

Each domain model object to be shown in a TreeView is finally wrapped in an instance of class *TreeNode*. The object wrapped by a TreeNode is called a tree node's **subject**. A TreeNode keeps track of an entry's expand/collapse status and caches the children once fetched by a children block. It also provides various options to additionally describe an entry. These options allow to specify:

- The type of node (is used to determine the icon to display).
- A specific icon for this node (overrides default icon detection via type).
- A specific display string to use as the entry's label.

- Information about whether children can be fetched or not.

When a TreeAdaptor automatically wraps entries fetched through a children block with instances of TreeNode, these tree nodes are initialized with default values, which are:

- type of node: *#folder*
- display string: the entry's *displayString*
- icon: determined according to the node's type
- has children: *true*

Variables

subject *<Object>*

The real hierarchy entry wrapped by the TreeNode. Can be accessed via *subject* but not changed. A TreeNode is initialized with a certain subject on instance creation using the instance creation methods *for: aSubject* or *for: aSubject label: aString*.

type *<Symbol* **|** *Object>*

The type of node. Can be accessed via *type* and *type:*. A node's type is used to determine the icon to display for that node in a tree view. The type value is used as a key in the TreeView's image list dictionary.

There are two pre-defined types *#folder* and *#leaf*. Type *#folder* is the default type of each TreeNode as being initialized by TreeNode's instance creation methods (*for:label*). It is also assigned to variable *type* if you send an instance the message *isParent: true*. Type *#leaf* will be used if you send the message *isLeaf: true* (or *isParent: false*, resp.).

An application can initialize TreeNodes with user defined types. A type can generally be any kind of Object, however, most often it is a Symbol.

displayString *<String>*

The string to display as the node's textual label in a tree view. Can be accessed via *displayString* and *displayString:*. There's also a special instance creation method *for:label:* provided, that allows you to specify the label string to use.

If you don't set an instance's display string explicitly the variable is initialized from the subject's displayString.

icons *<nil or Association<Image -> Image>>*

The icons to display to the left of the node’s textual label in a tree view in collapsed and expanded state. Can be accessed via *icons* and *icon:* or *icons:*.

Initially this variable is *nil* for a newly created instance. In this case the TreeView takes the icons to display from its image list using the node’s type to select the corresponding icon. If you assign the icons to use explicitly, this overrides the image list entry for this node.

parent *<TreeNode or nil>*

The node’s parent node. This variable is automatically assigned when children are fetched. You may not change it manually. Can be accessed via *parent*. (You can access an entries parent entry using something like *(aTreeAdaptor nodeFor: anEntry) parent subject*).

children *<#none or nil or Collection<TreeNode>>*

The node’s children. Can be accessed with *children* and *children*:. Depending on the value of this variable the instance is in one of these states:

#none	the node definitely has no children; it is a leaf
nil	the node might have children; but children have not yet been fetched for this node
empty collection	children have been fetched but no children have been found
collection	children have been fetched and found

Normally a node’s children are fetched by evaluation of a TreeAdaptor’s children block. However, you can as well assign a node’s children statically.

Initially this variable is *nil* for a newly created instance. This means that the node might have children but this is not sure, since there hasn’t yet occurred a fetch for children for this node. When children have been fetched for a node, this variable holds a collection of the children found. However, this collection can be empty.

You can explicitly specify that an instance definitely has no children by sending it *hasChildren: false*. This sets *children* to *#none*.

You can check if an instance has children with *hasChildren* or *definitelyHasChildren*. Only if the latter returns *true* should you access the contents of *children*.

status *<#collapsed or #expanded>*

The node’s current status in the tree view. Can be accessed via *status* but not changed explicitly. This is done internally. Initially, the status of a newly created instance is *#collapsed*.

Instance Creation (class)

for: *anObject*
for: *anObject* ***label:*** *aString*

Create a new instance with subject assigned to *anObject*. Variant 2 additionally sets the new instance's display string to *aString*. Further instance variables are initialised with default values.
^ *<TreeNode>*

anObject ***asTreeNode***

This is a convenient method added to class *Object* that is implemented as *^TreeNode for: self*. This method is overridden in TreeNode as *^self*.
^ *<TreeNode>*

Initialisation

hasChildren: *aBoolean*

Specify in advance whether the receiver has children or not. If aBoolean is *false*, instance variable *children* will be set to *#none* and no children block will ever be evaluated for this node.

isParent: *aBoolean*

The same as *hasChildren:* except that the node’s type is set to *#folder* if the parameter is *true* and to *#leaf* if it is *false.*

isLeaf: *aBoolean*

Convenience for as *isParent: aBoolean not.*

Accessing

children

Get the collection of children to the receiver’s subject.
^ ***children*** *<#none or nil or Collection<TreeNodes>>*

children: *aCollectionOrNil*

Set the collection of children explicitly. The parameter must be either nil or a collection. If a collection is given, this will ensure, that now children will be fetched for this node using a children block. If the parameter is nil, this will lead to the children block being evaluated for this node the next time it is expanded.

The elements in the collection may either be instances of TreeNode or any other kind of objects. If the elements are no TreeNodes, they are supposed to be plain entries of a hierarchy and are wrapped by automatically created instances of TreeNode.

displayString

Get the string to display as the node's textual label in a tree view. If the display string has not been initialized explicitly on node creation, this method returns *subject displayString* as a default.
^ ***displayString*** **|** *subject displayString <String>*

displayString: *aString*

Set the string to display as the node's textual label in a tree view.

icons

Get the association of icons to display to the left of the node's textual label in a tree view.
^ ***icons*** *<nil or Association>*

icons: *anAssociation*

Set the icons to display to the left of the node's textual label in a tree view. The parameter is an association with the key specifying the icon to display in collapsed state, the valie specifying the icon to display in expanded state.

icon: *anImage*

Convenience for: *aTreeNode icons: anImage* -> *anImage*, where the same image shall be used in both, collapsed and expanded state.

type

Get the node's type.
^ ***type*** *<Object>*

type: *anObject*

Set the node's type.

parent

Get the receiver's parent node.
^ ***parent*** *<TreeNode>*

path

Answer a collection with all the subjects of the receiver and all its parent nodes.
^ *<Collection<Object>>*

nodesPath

Answer a collection with the receiver and all its parent nodes.
^ *<Collection<TreeNodes>>*

nodesPathString
nodesPathStringWith: *aDelimiterCharacter*

Returns a string representation of the *nodesPath* with each path component being resolved from the respective node's displayString, seperated by a delimiter character. The default delimiter character as being used in variant 1 is *$.*.
^ *<String>*

status

Get the node's current status.
^ ***status*** *<#collapsed or #expanded>*

subject

Get the node's subject.
^ ***subject*** *<Object>*

Testing

= *aNodeOrAnObject*

Compares the receiver to the parameter based on **equality comparison**. Returns *true* if either *aNodeOrAnObject* is a domain model object and is **equal** to the receiver's subject, or if it is the receiver itself.
^ *<Boolean>*

<= *anotherNode*

Compares the display string of two nodes with <=. Allows instances to be inserted in SortedCollections with default sortBlocks. It is implemented as: ***displayString <= another displayString*** *<Boolean>*

hasChildren

Check whether the receiver has children. This returns *true* either when the instance variable *children* holds a non-empty collection of children, or when it is nil. The latter means that the node might possibly have children but no children have been fetched for this node yet.
^ *<Boolean>*

definitelyHasChildren

Check whether the receiver has children. This returns *true* only when the instance variable *children* holds a non-empty collection of children. On contrary to *hasChildren*, this method also returns false if *children* is nil, that is when no children have been fetched yet.
^ *<Boolean>*

isCollapsed

Convenience for:
^ ***status == #collapsed*** *<Boolean>*

isExpanded

Convenience for:
^ ***status == #expanded*** *<Boolean>*

isTreeNode

This method overrides the corresponding method in class *Object* to always return *true*. (The implementation in *Object* always returns *false*)
^ ***true***

Class IdentityTreeNode

Inherits from: *TreeNode*

This is a subclass of *TreeNode* that allows to adapt to hierarchies based on object identity comparisons rather than object equality.

Testing

= *aNodeOrAnObject*

Compares the receiver to the parameter based on **object identity**. Returns *true* if either *aNodeOrAnObject* is a domain model object and is **identical** to the receiver's subject, or if it is the receiver itself.
^ *<Boolean>*

Class MultipleParentTreeAdaptor

Inherits from: *TreeAdaptor*

This class provides some basic-level support for adapting to multiple-parent hierarchies. It basically cares for correct detection and manipulation of each instance of *TreeNode* for a given domain model object in the common operations, such as on adding or removing nodes.

The basic difficulty when dealing with multiple-parent hierarchies in a TreeView is, that a certain entry may be displayed several times in the TreeView. This is because if a node has more than one parent, it would have to be displayed in each of these parents' children branches. In result, there may be **more than one instances** of *TreeNode* for a certain domain model hierarchy entry.

Accessing

allIndexesOf: *aNodeOrAnObject*

Looks up all the nodes for *aNodeOrAnObject* and returns a collection of according indexes.
^ *<Collection of Integers>*

allNodesFor: *aNodeOrAnObject*
allNodesFor: *aNodeOrAnObject* ***ifNone:*** *aBlock*

Looks up all the nodes for *aNodeOrAnObject* and returns a collection of the matching nodes.
^ *<Collection of TreeNodes>*
! *notFoundError* when no entry was found (only in variant 1).

Modifying

add: *aNodeOrAnObject* ***asChildOf:*** *aParentNodeOrObject*

Adds new TreeNodes to all the TreeNodes corresponding to the specified parent. If the parent parameter is a TreeNode, a new node will be added to this very TreeNode only. Otherwise, a new node will be added to all the TreeNodes having parent as the subject.

This means, in order to add a new node to all existing TreeNodes for a parent entity in your domain model, you should provide the parent entity as the parent parameter and not a parent TreeNode.

E.g.:

aTreeAdaptor ***add:*** *'New Entry'* ***asChildOf:*** *aSelectionInTree selectedNode*.

will add a new TreeNode for: *'New Entry'* to only the very TreeNode returned by selectedNode.

aTreeAdaptor ***add:*** *'New Entry'* ***asChildOf:*** *aSelectionInTree selection.*

will add a new TreeNode for: 'New Entry' to any TreeNode, that has the object returned by selection as its subject.

remove: *aNodeOrAnObject* ***fromParent:*** *aParent*
remove: *aNodeOrAnObject* ***fromParent:*** *aParent* ***ifAbsent:*** *aBlock*

Removes the denoted object from all the TreeNodes corresponding to the specified parent. If the object parameter is a TreeNode, only this very TreeNode will be removed. Otherwise, all the TreeNodes whose subject equals the object in question will be removed.

If the parent parameter is a TreeNode, children will be removed from this very TreeNode only. Otherwise, children will be removed from all the TreeNodes having parent as the subject.

E.g.:

```
aTreeAdaptor remove: aSelectionInTree selectedNode
        fromParent: aParent.
```

will remove the very TreeNode returned by *selectedNode* only.

```
aTreeAdaptor remove: aSelectionInTree selection
        fromParent: aParent.
```

will remove any TreeNode, that has the object returned by *selection* as its subject.

```
aTreeAdaptor remove: something
        fromParent: aSelectionInTree selectedNode.
```

will remove something from the very TreeNode returned by *selectedNode* only.

aTreeAdaptor ***remove:*** *something*
fromParent: *aSelectionInTree selection.*

will remove something from any TreeNode, that has the object returned by *selection* as its subject.

! notFoundError when no entry was found for aParent
(only in variant 1).

removeFromParent: *aParent*

This is a convenient method for:

aTreeAdaptor ***remove:*** *aTreeNode subject*
fromParent: *aTreeNode parent*

I.e. it expects a TreeNode as its parameter and will remove any TreeNode referring to the same subject from all TreeNodes referring to the same parent.

Class TreeView

Inherits from: *SequenceView*

Class TreeView basically provides the functionality for displaying hierarchical structures. It inherits from its superclass the basic behavior of list views. As its model's value however, it expects an instance of class *TreeAdaptor*. This tree adaptor is stored in instance variable *sequence.*

Each entry in a tree view can be displayed with an associated icon. Moreover, a different icon can be displayed when an entry is expanded. Generally, which icon will be displayed is determined by the nodes type. Each TreeView can therefore be equipped with an image list, a dictionary of icons with the types of nodes as the dictionary keys. If no image list is specified explicitly, TreeView uses a default image list. This default image list has entries for the pre-defined types *#folder* and *#leaf*.

Variables

imageList *<nil or Dictionary>*

A dictionary with entries specifying the icons to display for different types of entries. The dictionary keys are type of nodes which will be added to the hierarchy (see *TreeNode type*). The entries' values are either a single image or an Association of two images. If there is only one image specified for a type, this image will be used in both collapsed and expanded state of an entry. If there is an association of two images given for a type, the first one (key) will be used for entries in collapsed state and the second one (value) for entries in expanded state.

If *imageList* is nil, no images will be displayed in front of the entries' labels.

displayLines *<Boolean>*

Denotes whether the view displays the thin lines connecting the entries in the hierarchy. Can be accessed via *displayLines* and *displayLines:*.

displayButtons *<Boolean>*

Denotes whether the view displays the expand/collapse buttons ([+], [–]) in front of each entry. Can be accessed via *displayButtons* and *displayButtons:*.

leftOffset *<Integer>*

Private

Accessing

imageList

Get the dictionary of images to be used for different types of entries / nodes.
^ *imageList* *<nil or Dictionary>*

imageList: *aDictionary*

Set the dictionary of images to be used for different types of entries / nodes.

displayImages

Test whether the view is configured to display images in front of the entries' labels. The implementation is:
^ imageList notNil

displayImages: *aBoolean*

Set whether the view should display images in front of the entries' labels. The implementation impacts instance variable *imageList*. If *aBoolean* is true and no image list has been provided yet, the *imageList* is set to the class's default image list (*TreeView class>>defaultImageList*). If *aBoolean* is false, *imageList* is set to *nil*.

displayButtons

Test whether the view is configured to display expand/collapse buttons in front of the entries' labels.
^ ***displayButtons***

displayButtons: *aBoolean*

Set whether the view should display expand/collapse buttons in front of the entries' labels.

displayLines

Test whether the view is configured to display thin lines in front of the entries' labels.
^ ***displayLines***

displayLines: *aBoolean*

Set whether the view should display thin lines in front of the entries' labels.

Class Initialisation (class)

displayOpenFolderWhenSelectedOnly

Answers the value of a corresponding class variable that tells whether the *open icon* (e.g. an open folder) is displayed for expanded nodes or for the selected/opened entry only. The pre-configuration has this variable set to *true*. However, you can still restore the old behaviour by evaluating: *TreeView displayOpenFolderWhenSelectedOnly: false*.
^ *<Boolean>*

displayOpenFolderWhenSelectedOnly: *aBoolean*

Change the value of the corresponding class variable (see above).

Constants (class)

defaultImageList

Get the class's default image list dictionary
^ ***DefaultImageList*** *<Dictionary>*

defaultIcons

Get the Association of icons (key = collapsed icon, value = expanded icon) which are used, when no entry can be found in a TreeView's current image list for a node. This is when there's no entry in the image list corresponding to the node's type.
^ ***DefaultImageList at: #folder*** *<Association>*

folderIcon

Get the default icon which is used for nodes with the pre-defined type *#folder* in collapsed state.
^ *<Image>*

openFolderIcon

Get the default icon which is used for nodes with the pre-defined type *#folder* in expanded state.
^ *<Image>*

leafIcon

Get the default icon which is used for nodes with the pre-defined type *#leaf* in collapsed and expanded state.
^ *<Image>*

Index