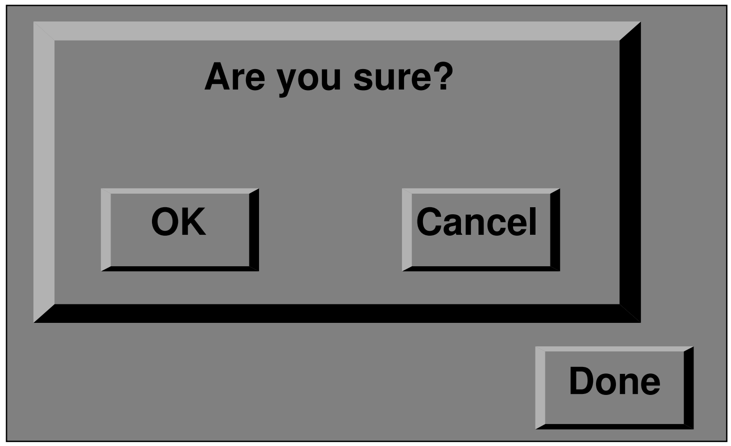
Hierarchy

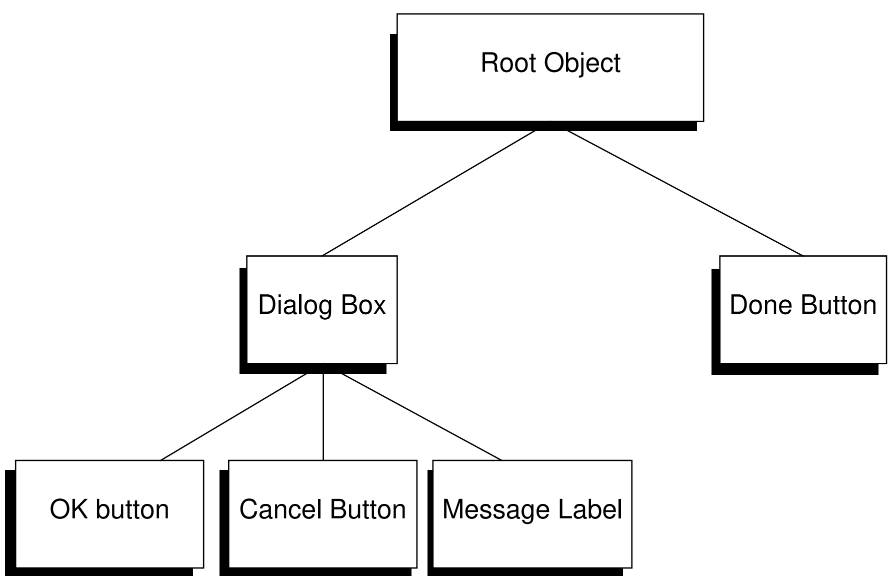
Introduction: Parents and Children

SUIT objects can be composed of other SUIT objects for purposes of creating more complicated widgets such as dialog boxes:



In the above example, we have a dialog box widget that contains a label widget and two button widgets. We say that the label and two buttons are the *children* of the dialog box and that the dialog box is the *parent* of the label and two buttons.

All objects are ultimately the children of an invisible *root object*. This relationship we can draw as a tree:



In this section, we will discuss how hierarchical widgets are different than the simple "flat" widgets that you might be used to, and what you need to do to compose a hierarchical widget of your own. We will start by looking at the basic building block of hierarchial widgets: the bulletin board widget.

Bulletin Boards

Bulletin boards are a class of widget that can hold other widgets. Bulletin boards can even contain other bulletin boards, allowing you to create very complex widgets. When clicked on, a bulletin board will pass the event down to its children and when moved, the bulletin board will make sure that its children are moved as well. Composing widgets with bulletin boards is easy: for example, to create a bulletin board that contains a button, you use the following code:

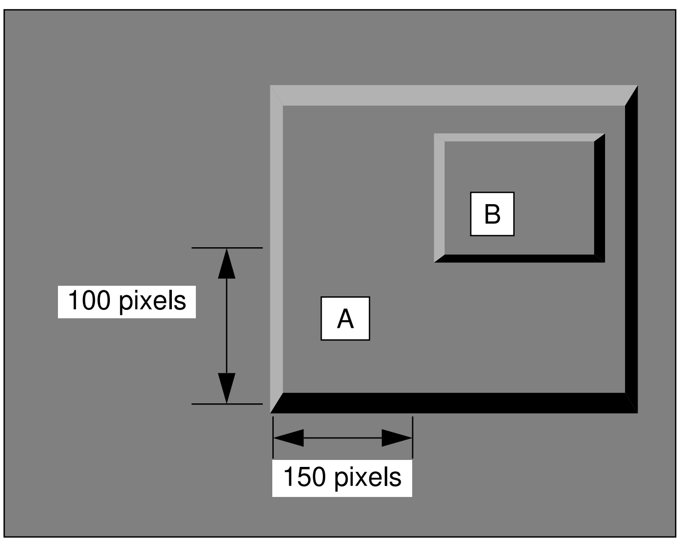
```
board = SUIT_createBulletinBoard("sample");
button = SUIT_createButton("my button", myCallbackFcn);
SUIT_addChildToObject (board, button);
```

Examples of bulletin board creation can be found in the collection of example files that comes with the SUIT installation. The composition calls that you see here do not specify *where* the label and the button will appear in the bulletin board, all they do is make sure that the button and label are treated as children of the bulletin board. To place the label and button in a particular position inside the bulletin board, you need to set the VIEWPORT property of the children.

Viewports

Viewports for hierarchical widgets are based relative to the lower left hand corner of the *parent widget*, not the lower left hand corner of the screen.

A is a hierarchical widget with one child, a button B.



Widget B's viewport is measured from the lower left hand corner of widget A, not the lower left hand corner of the application window.

If B is 50x50 pixels, then B's viewport is lower left = (150, 100) upper right = (200, 150).

Widget A's viewport is measured from the lower left hand corner of the application window.

Changing Childrens' Viewports

There are two ways of arranging children inside a parent widget:

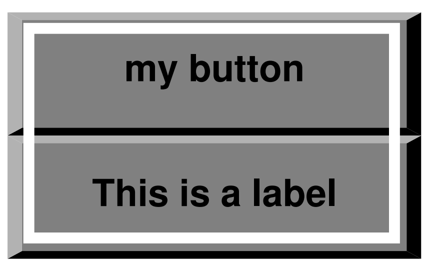
- Interactively arrange the children using SUIT-move
- Explicitly arrange the children in C code by setting viewport properties.

Of course, you can't just use SUIT-move to move a bulletin board's children, the SUIT-move moves the bulletin board as a whole. In order to "get inside" a hierarchical widget, you need to open it up with SUIT-o. When opened, a red border will appear around the bulletin board so that the children can be accessed with SUIT commands like SUIT-click SUIT-e, etc. You can even add and remove children from a bulletin board by interactively dragging the objects into and out of the bulletin board.

The second way of laying out the bulletin board's children is by changing their viewports with calls to `SUIT_setViewport()`. This method is more precise, but somewhat more time consuming. The following code places a button and a label inside the parent widget:

```
SUIT_changeObjectSize(board, 400, 200);

SUIT_setViewport (label, VIEWPORT,
                  SUIT_defViewport(0, 0, 400, 100));
SUIT_setViewport(button, VIEWPORT,
                  SUIT_defViewport(0,100, 400, 200));.
```



Viewport of button	(0,100, 400, 200)
Viewport of label	(0, 0, 400, 100)

When you type SUIT-o,a border appears around the widget, letting you know that the widget has been "opened" and that the widgets inside can now be moved interactively.

NOTE: It is important that these calls come AFTER the call to `SUIT_addChildToObject()`. Because viewports are relative to the parent, if you set the viewport of the button before adding it to the bulletin board, it will compute its location relative to the root object, not the bulletin board.

Setting Viewports Using GP Coordinates

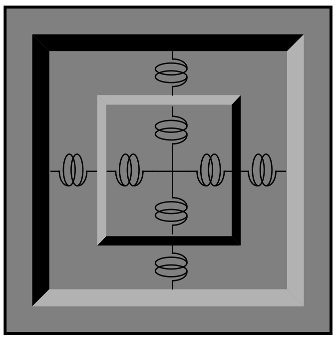
`SUIT_mapToParent()` is a special mapping function that allows you to use GP coordinates to position a child widgets inside its parent. The function takes an object and four floating point numbers that represent the corners of a GP_rectangle and returns the unique SUIT_viewport inside the parent that corresponds to the given GP_coordinates. For example, to place the above label inside the bulletin board as shown, you would use this code:

```
SUIT_addChildToObject (board, label);
SUIT_setViewport(label, VIEWPORT,
                  SUIT_mapToParent(label, 0,0, 0,0, 1.0, 0.5));
```

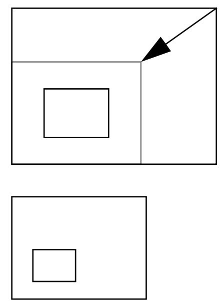
This mapping function makes it possible to place and size widgets using the same coordinate system used by the GP graphics calls to draw lines and such. You can see more examples of this mapping call in the example programs that come with the SUIT distribution.

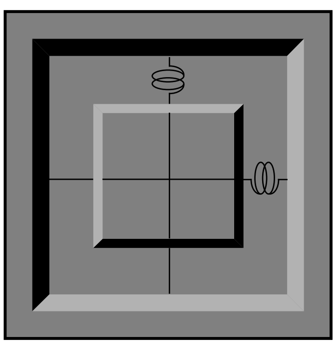
Springiness

Suppose you resize our example bulletin board to be smaller. Should the button and label get proportionally smaller or should they stay the same size as they were before the resize? The answer lies in the *springiness* of the label and the button, a property that allows widgets to specify how to resize if their parent changes in size. The metaphor we use here is one of *springs* and *stiff bars*: imagine that each widget is connected to the four walls of its parent by either springs or stiff bars. **The rule to remember is:** As the parent's walls move, the widget will stay the same distance from a wall if it is connected by a stiff bar. In the same fashion, widgets describe whether they are allowed to resize by having either a stiff bar or a spring in each of the two possible directions, horizontal and vertical.

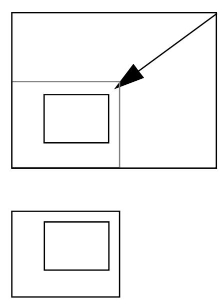


This is the default value for springiness. With no stiff bars, the child resizes proportionally as its parent resizes.





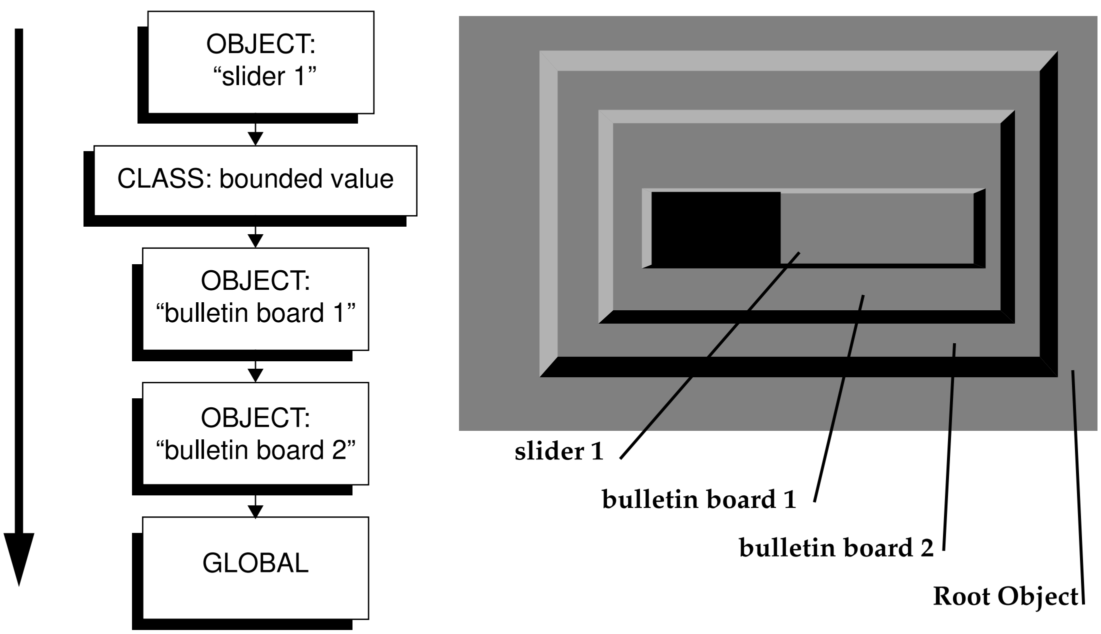
Here, springiness is set so that the child will always remain the same distance from the lower left hand corner of the parent, but the distance to the upper right hand corner is allowed to float. Because there are bars on the inside of the widget, both horizontally and vertically, the widget will stay the same size, regardless of how its parent resizes.



Though it may not be obvious at first, it turns out that there must be *some* springiness vertically and horizontally. If a widget was connected to its parents by a stiff bar on both its left and right sides, **and** had a stiff horizontal bar inside, resizing the parent would require following conflicting orders: the bars on the outside tell the child's left and right sides to stay the same distance from the parent's sides, but the stiff bar inside tells the child widget not to get wider. You might say that the child widget is "overconstrained." SUIT's springiness widget (above) does not allow you to select these illegal springiness states.

Property Lookup Order

Property lookup on a hierarchical widget is done in a slightly different order from the lookup on “flat” objects. The lookup begins the same: it starts at the OBJECT level of the widget in question and if the property lookup fails, it then goes to the appropriate CLASS level. Failure to find a property at the CLASS level would usually mean looking for a property at the GLOBAL level, but with a hierarchical widget, property lookup instead continues at ***the object level of the parent***. If you still fail to find the property, the lookup continues at the OBJECT level of the next immediate parent (its grandparent), until you reach the root object, where a lookup at the GLOBAL level takes place. For example, imagine an object called “slider 1” which is a child of “bulletin board 1” which in turn is a child of “bulletin board 2” which is a child of the root object. The property lookup would go like this:



Registering a Bulletin Board as a New Class

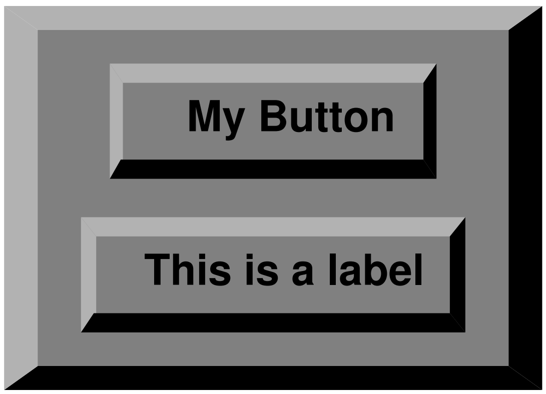
Bulletin boards belong to the “bulletin board” class of widgets. If you set a property at the “bulletin board” class, all bulletin boards will inherit that property, if they don’t specify that property at the OBJECT level. This can present a problem. Because bulletin board widgets are used to create new widgets, it is often useful to associate a new class name with a widget created out of a bulletin board. For example, suppose you had created a hierarchical widget using a bulletin board: a data entry form with several type in boxes and an OK button. From your point of view, you’ve created a completely different kind, or class, of widget: the “data entry form” class. To SUIT, though, it’s just a bulletin board. Properties set at the “bulletin board” class (such as foreground color) would affect bulletin boards AND your data entry form. To allow you to set different class level properties on these two kinds of widgets, you need a way of telling SUIT that the bulletin board you are creating is not really a bulletin board for the purposes of class level property lookups, but is some other class instead. The SUIT call that does this is:

```
form=SUIT_createBulletinBoardWithClass("form1","data entry form");
```

Where “data entry form” is the new class name you want to use for property lookups.

Shrink To Fit

The boolean property SHRINK_TO_FIT that governs labels, buttons and type in boxes causes a widget's viewport to shrink around the size of the text in the LABEL property, plus whatever MARGIN there is. It is not possible to interactively resize a widget that obeys the SHRINK_TO_FIT property. The viewport of the widget shrinks about its center point, so the four sides of the widget move, but the center of the widget remains unchanged. In our example, if SHRINK_TO_FIT is set to true for both the button and label, these widgets would assume their set locations inside the bulletin board and then would resize about their respective centers to create a widget that looked like the one shown below. (The label widget has had its HAS_BORDER property changed to TRUE for purposes of illustration).



Here, the SHRINK_TO_FIT property is toggled to TRUE for both the label and the button, making their viewports as small as possible, while still being able to see the text.

Notice that their viewports have shrunk about the center of the widget.

Stacker Widgets

There is a special hierarchical widget called a "stacker" that comes in two display styles, horizontal and vertical. These are essentially bulletin boards that perform some very simple geometry management by arranging their children to be packed vertically or horizontally. They are like all hierarchical widgets with respect to property lookups and children.

Employees

Employees are children that are registered for a particular display style. A good example of the use of employees is the bounded value widget. The scrollbar display style uses an elevator widget and two arrow button widgets. Clearly, these widgets need not exist for any other display style other than the scroll bar, so instead of children, we make these widgets employees of the scroll bar display style. For purposes of property lookup and composition, employees are treated exactly like children.