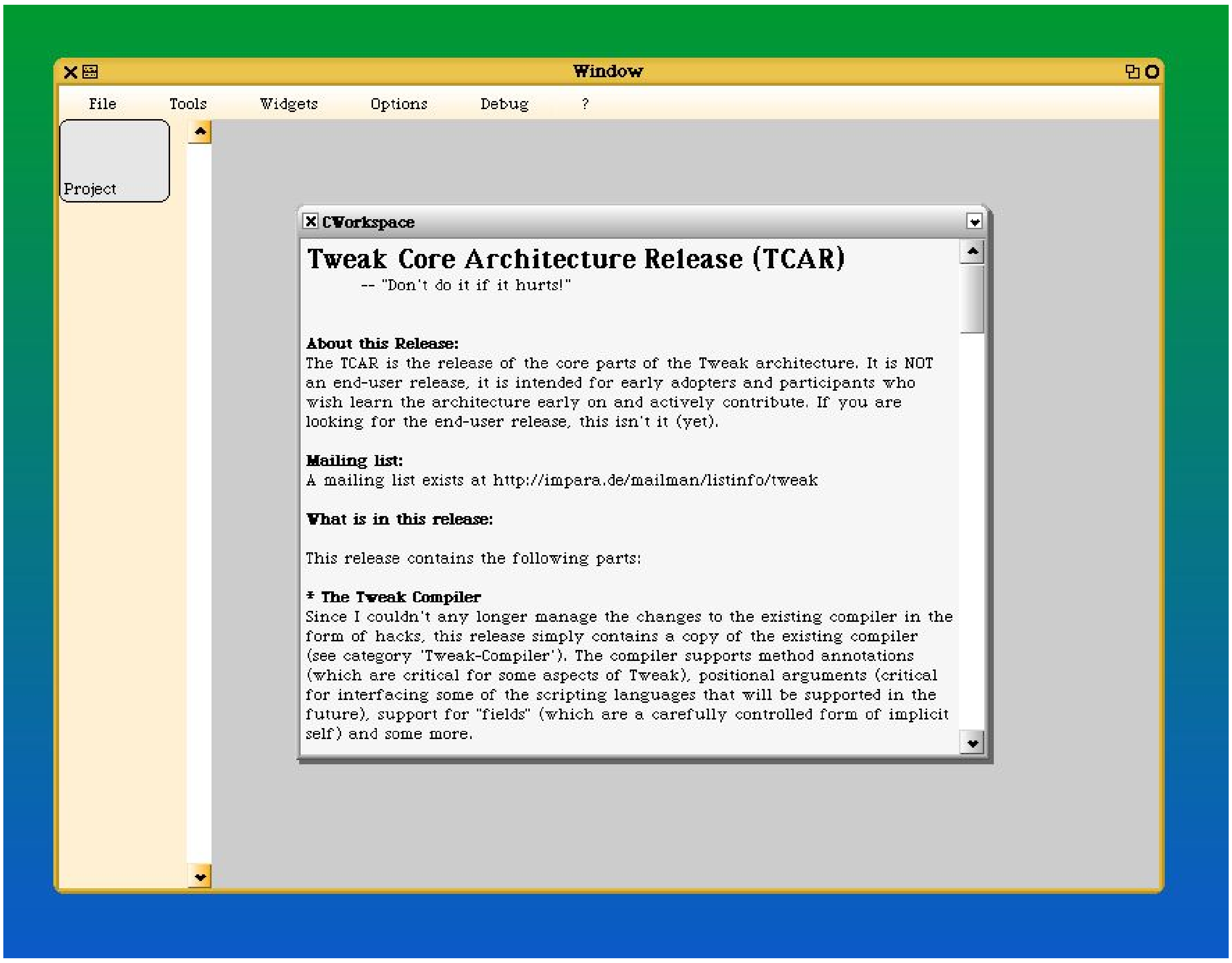
Short Introduction into Tweak

by Patty Gadegast, version 0.2, 07/14/2004

In General - What is Tweak ?

Tweak is a new user interface for Squeak. It is intended to replace the Morphic user interface in the future. Tweak objects can be created within the Tweak Project window only. Some standard widgets are already predefined and ready to use, i.e. buttons, text areas, scroll bars, as well as some familiar tools like a workspace and a browser, which are equivalent to the Morphic ones. However, not all development tools are recreated in the current version of Tweak yet, i.e. the debugger, therefore using Morphic is still necessary.



Tweak Core Architecture Release (TCAR)
-- "Don't do it if it hurts!"

About this Release:
The TCAR is the release of the core parts of the Tweak architecture. It is NOT an end-user release, it is intended for early adopters and participants who wish learn the architecture early on and actively contribute. If you are looking for the end-user release, this isn't it (yet).

Mailing list:
A mailing list exists at http://impara.de/mailman/listinfo/tweak

What is in this release:

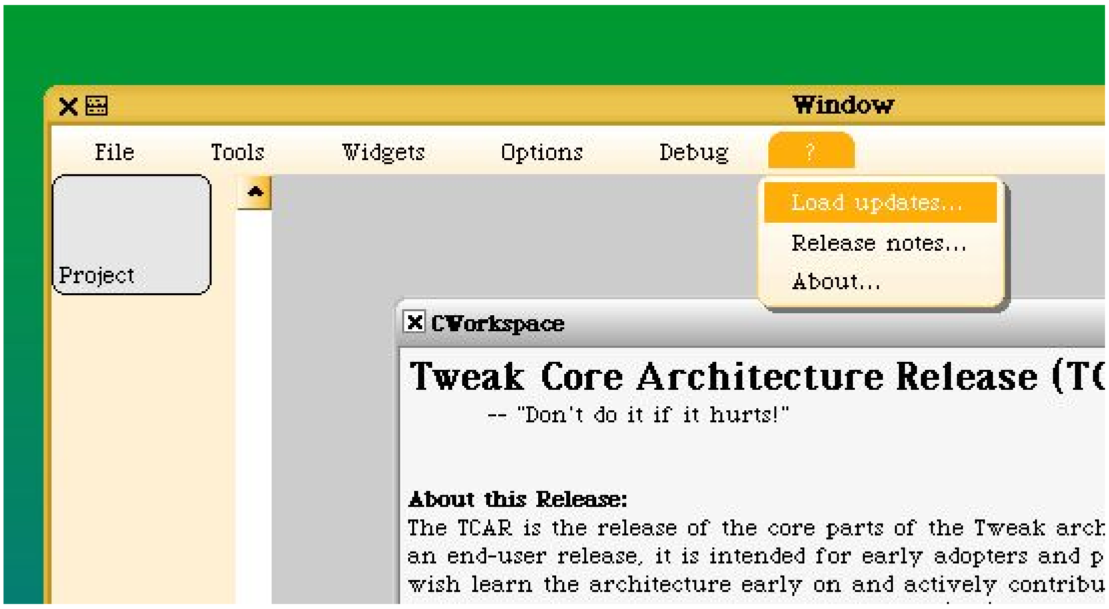
This release contains the following parts:

* **The Tweak Compiler**
Since I couldn't any longer manage the changes to the existing compiler in the form of hacks, this release simply contains a copy of the existing compiler (see category 'Tweak-Compiler'). The compiler supports method annotations (which are critical for some aspects of Tweak), positional arguments (critical for interfacing some of the scripting languages that will be supported in the future), support for "fields" (which are a carefully controlled form of implicit self) and some more.

Because the system is not finished yet, there still will be a lot of changes. For this reason, updating the system regularly is important.

Update the Tweak System

Basically, you need a Squeak image that includes Tweak, i.e. *Squeak3.x-TCAR.image*. You can open a Tweak project window (world menu – open – Tweak Project Window), within that window you are able to create and edit Tweak objects. To load updates use the item in the *help*-menu.



Tweak Core Architecture Release (T

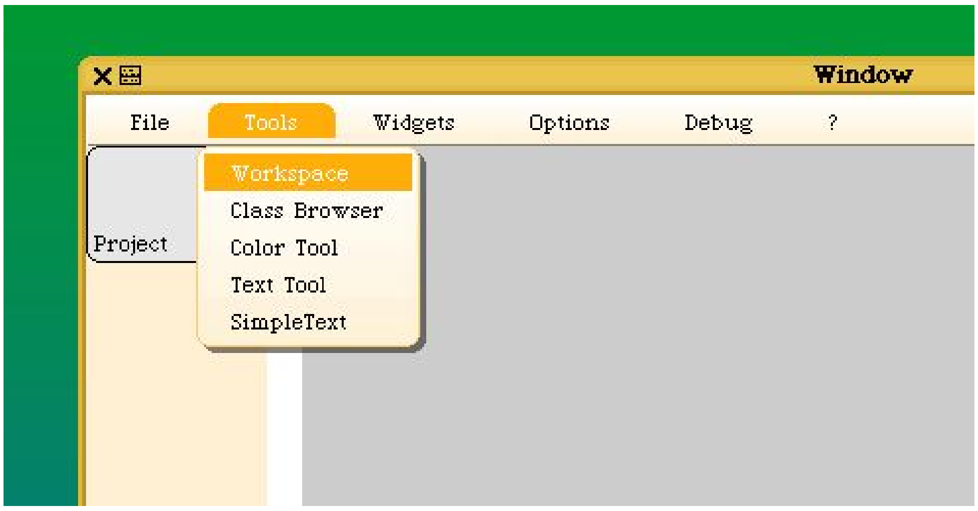
-- "Don't do it if it hurts!"

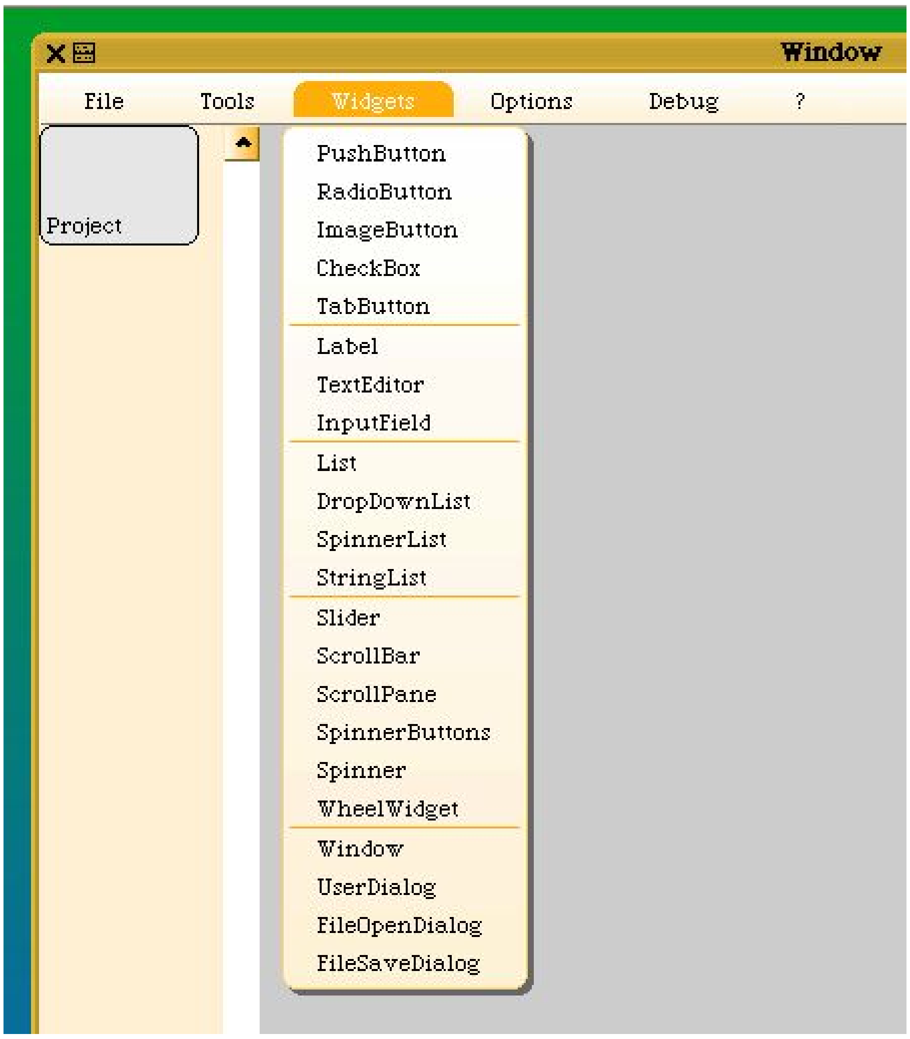
About this Release:
The TCAR is the release of the core parts of the Tweak arc
an end-user release, it is intended for early adopters and p
wish learn the architecture early on and actively contribu

Getting started – first steps in Tweak

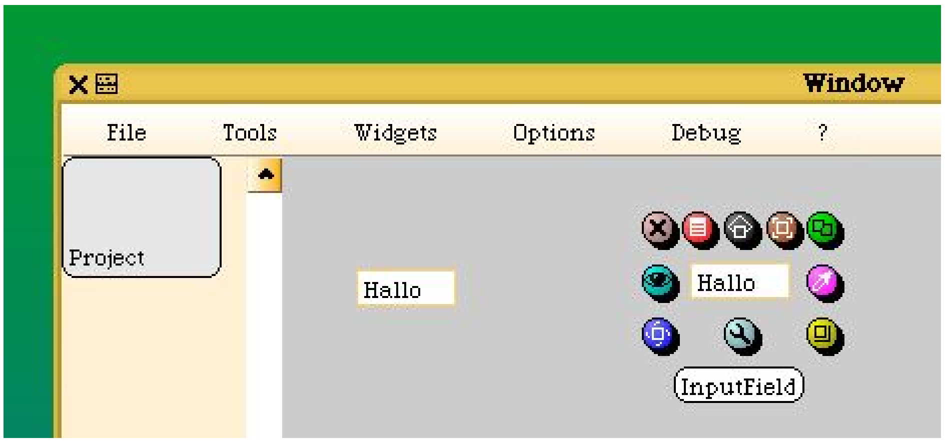
There are two ways to create objects with Tweak, either choose the widgets directly at the *widgets*-menu or create them by hand within the Tweak-*workspace*.

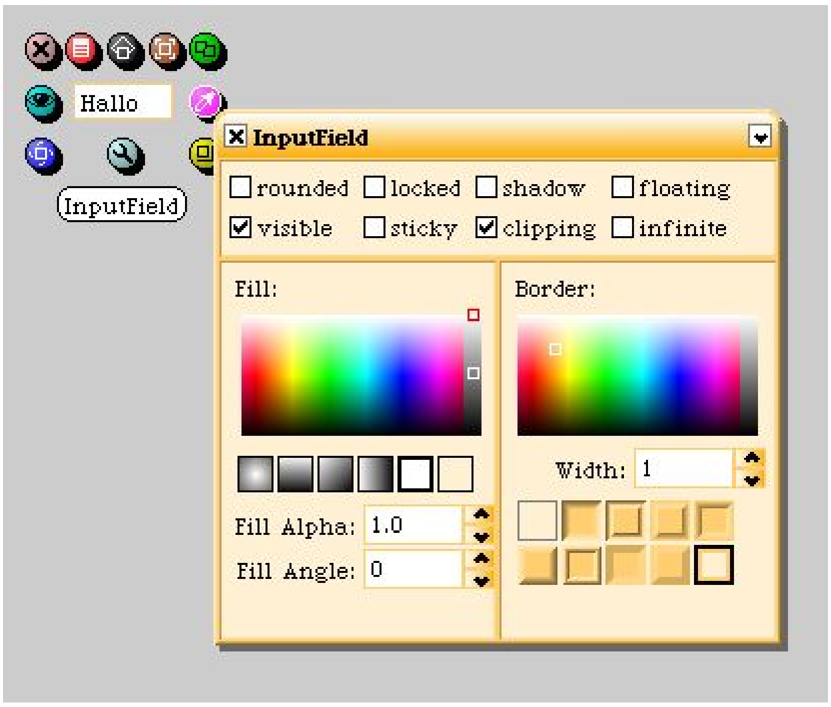
The *workspace* is opened by the menu *tools*. The Tweak-*workspace* works similary to the Morphic one, except using Tweak under Windows System, there the *alt*-key is used instead of the *crtl*-key for any shortcuts.





Standard widgets can be positioned anywhere in the Tweak project window. Using the Halos you can edit the objects, i.e. color, fillstyle, border.





Hallo

InputField

InputField

☐ rounded ☐ locked ☐ shadow ☐ floating

☑ visible ☐ sticky ☑ clipping ☐ infinite

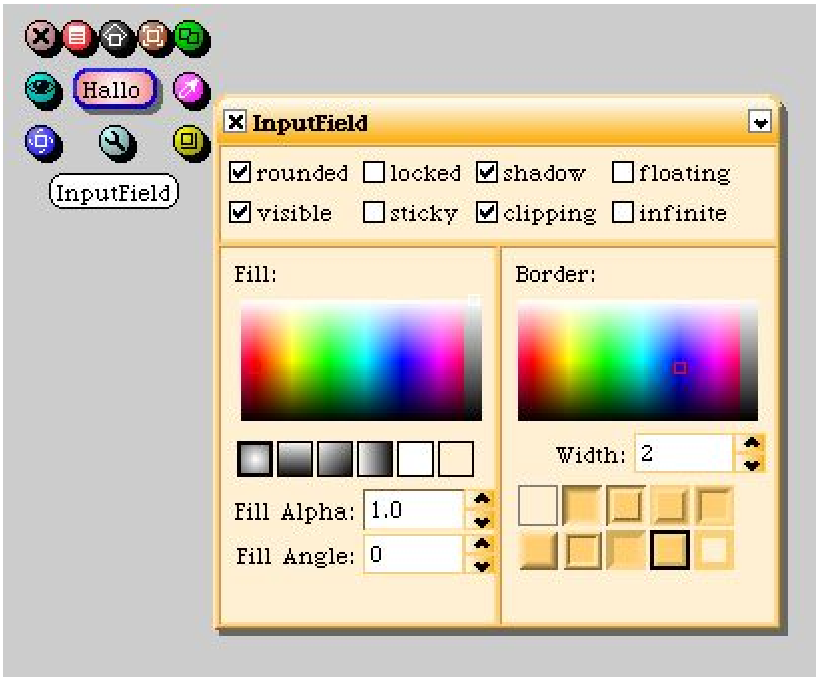
Fill:

Fill Alpha: 1.0

Fill Angle: 0

Border:

Width: 1



Hallo

InputField

InputField

☑ rounded ☐ locked ☑ shadow ☐ floating

☑ visible ☐ sticky ☑ clipping ☐ infinite

Fill:

Fill Alpha: 1.0

Fill Angle: 0

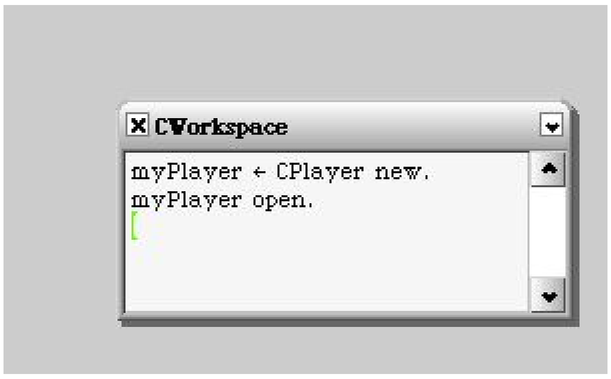
Border:

Width: 2

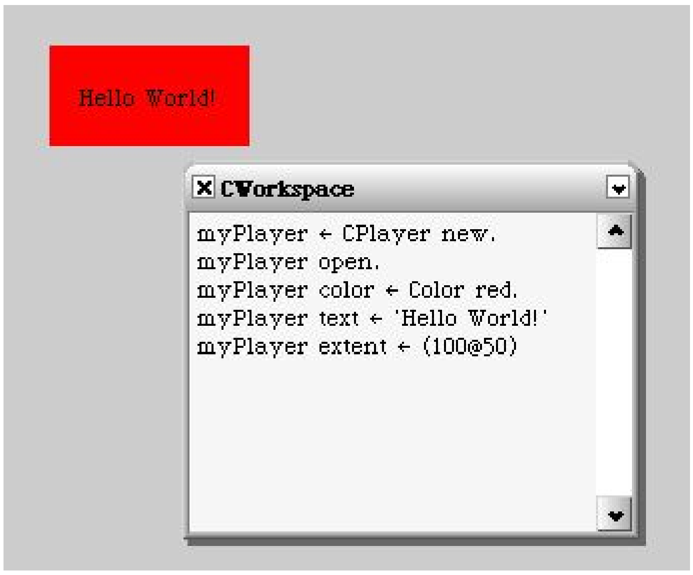
Creating objects by yourself within the workspace

You can create objects within the *workspace*. Therefor you have to create a new instance of *CPlayer*, generally it is tranparent. To make it visible, you have to assign a color. The newly created object is also editable by the halos.

Attention: after creating the object, it's present in cache, therefor you can access it by its name.



```
myPlayer ← CPlayer new.
myPlayer open.
```



```
myPlayer ← CPlayer new.
myPlayer open.
myPlayer color ← Color red.
myPlayer text ← 'Hello World!'
myPlayer extent ← (100@50)
```

The basis for all objects in Tweak is the *CPlayer*, the prefix C stands for Tweak. In the future there should be no prefixes, but that requires namespaces being established. Because the name *Player* is already used by another class in Squeak, the prefix is necessary up to now.

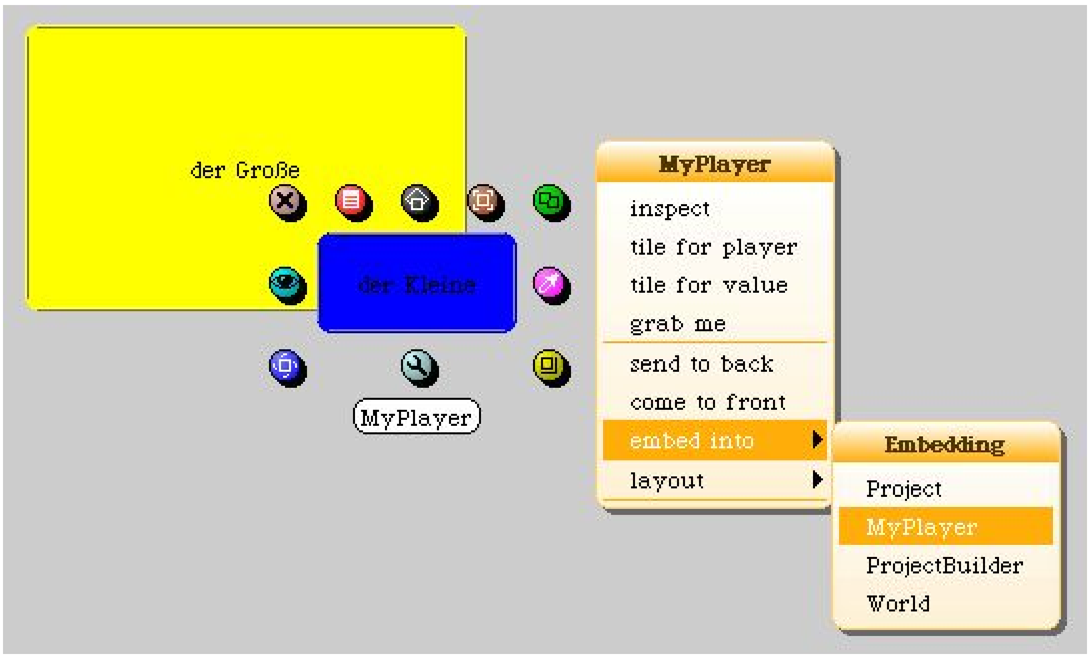
Some main attributes of CPlayer

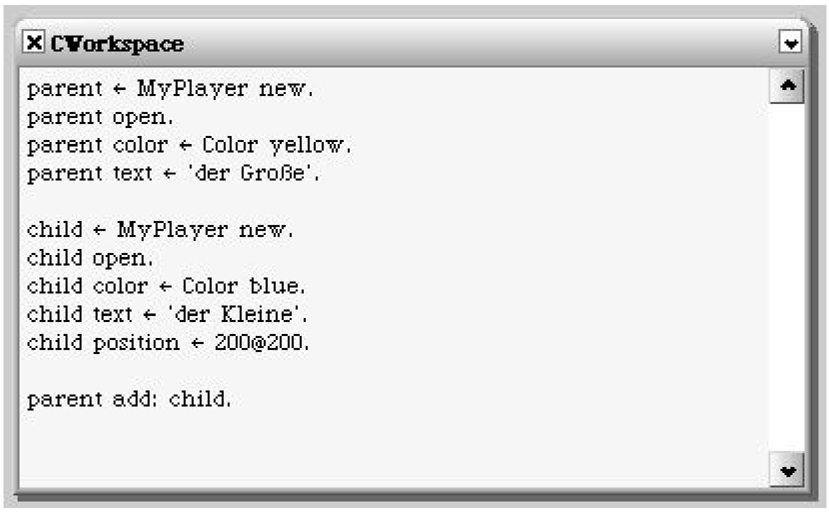
Aspect	**Attributes**	**Declaration**	**Allowed types**
General	roundCorners	Round up the corners	Boolean
	locked		Boolean
	visible	visible/invisible	Boolean
	sticky		Boolean
	clipping	Inner objects are clipped at the object bounds	Boolean
	floating		Boolean
	infinite	Object behaves like it's size were infinite	Boolean
Color	color		Color
	fillAlpha	Transparency of fill	Number
	fillAngle	Direction of fill	Number
	fillStyle	Style of fill	Symbol: #none, #solid, #horizontalGradient, #verticalGradient, #diagonalGradient, #linearGradient, #radialGradient
Border	border		Border
	borderColor	Color of border	Color
	borderWidth	With of border	Number
	borderStyle	Style of border	Symbol: #simple, #inset, #raised, #complexAltFramed, #complexAltInset, #complexAltRaised, #complexFramed, #complexInset, #complexRaised
Geometry	position		Point
	extent		Point
	bounds		Rectangle
	scale		Point
	angle	Angle of rotation	Number
	heading	Angle of rotation as vecto	Point (2d Vector)
	aspectRatio	Aspect Ratio	Number

Shadow	dropShadowColor	Color of shadow	Color
	dropShadowEnabled	Shadow behind object	Boolean
	dropShadowOffset	Displacement of shadow	Point
Text	text		String
	textColor		Color
	textFont	Font	Font
	textWrap	Wrap of text	Boolean
	textInset	Inset of text	Point
	textOffset	Offset of text	Point
	textAnchorPoint	Anchor point	Symbol: #center, #topLeft, #topRight, #bottomLeft, #bottomRight, #left, #right, #top, #bottom, #topCenter, #leftCenter, #bottomCenter, #rightCenter
	textBorderWidth	Border of text	Number
	textBorderColor	Color of textborder	Color
Graphic	graphic		Form
	graphicAnchorPoint		Symbol: #center, #topLeft, #topRight, #bottomLeft, #bottomRight, #left, #right, #top, #bottom, #topCenter, #leftCenter, #bottomCenter, #rightCenter
	graphicFit	Fit of graphic	Symbol: #rigid, #scale, #stretch, #tile, #autoScale
	graphicForm		Form
	graphicOffset	Offset topleft	Point
Layout	layout		Layout
	hResizing	Horizontal resizing	Symbol: #rigid, #shrinkWrap, #spaceFill
	vResizing	Vertical resizing	Symbol: #rigit, #shrinkWrap, #spaceFill
Scrolling	hScrollable		Boolean
	vScrollable		Boolean
	scrollingDisabled		Boolean

Linking and alignment of objects

Each *CPlayer* is also a container. Those containers are able to contain other *CPlayers*. You can add them either by menu or by workspace, calling the method *add:*. After that the added *CPlayer* is contained by the the other one, it will be moved and destroyed along with the parent object.



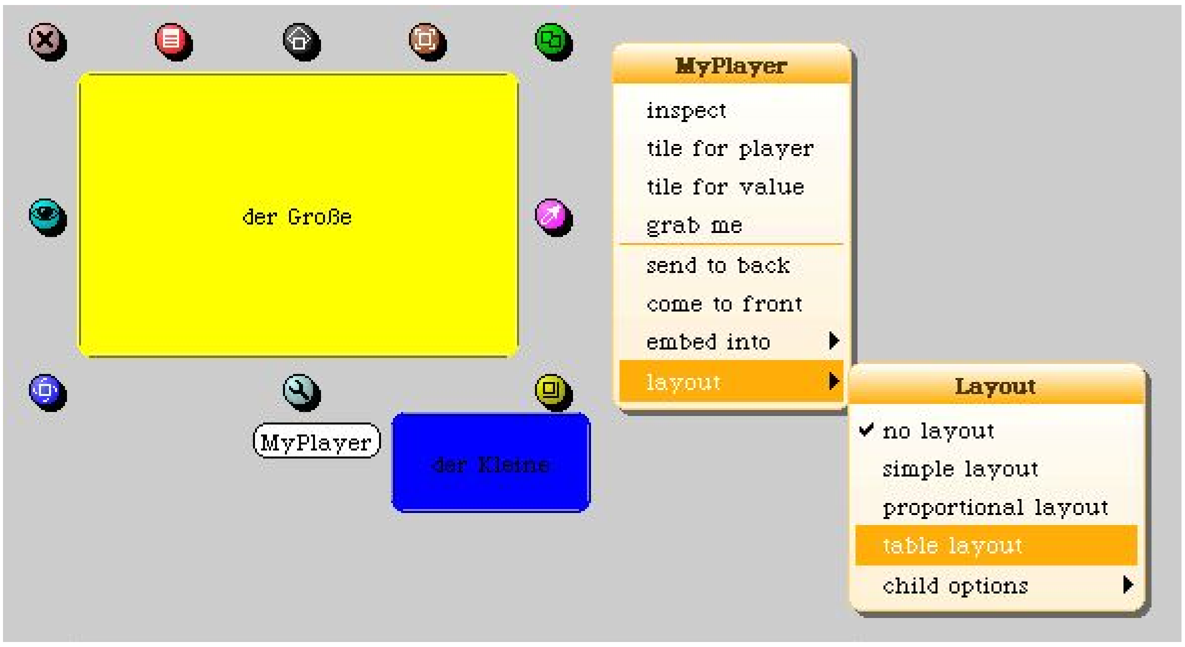


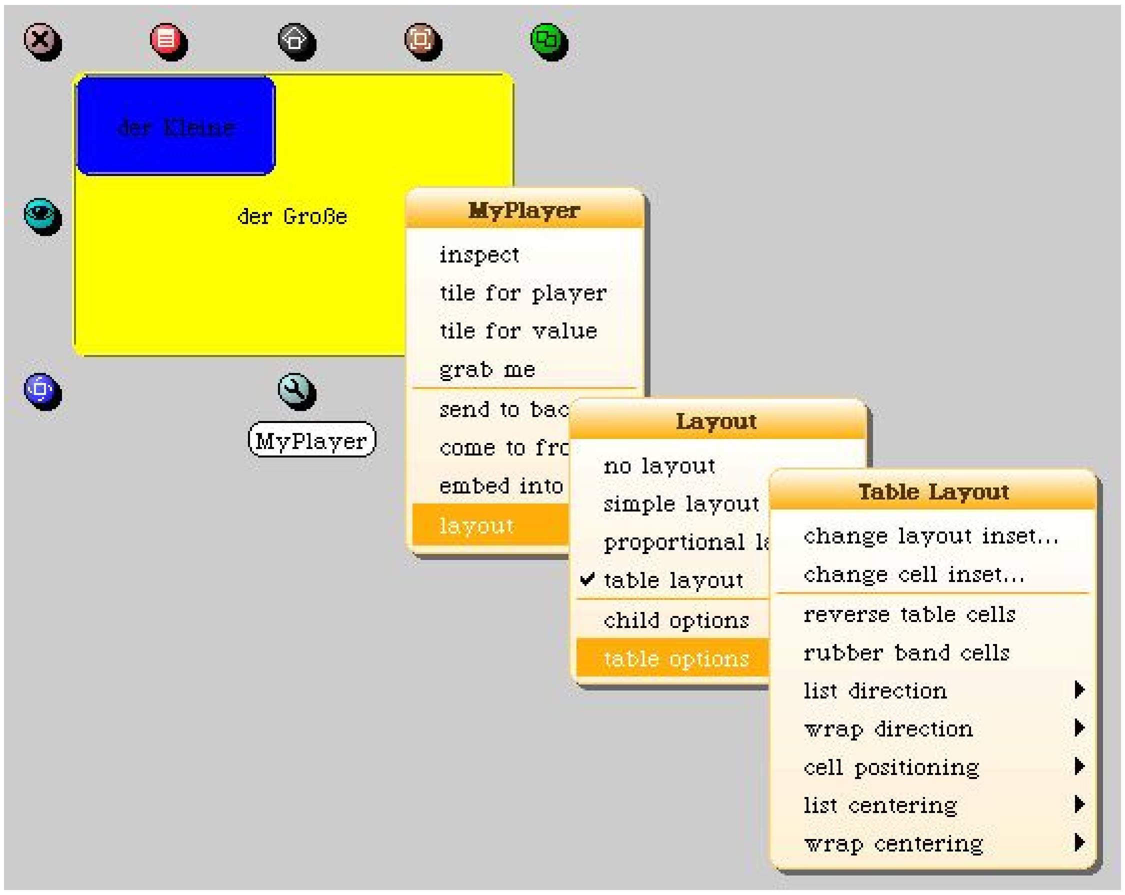
```
parent ← MyPlayer new.
parent open.
parent color ← Color yellow.
parent text ← 'der Große'.

child ← MyPlayer new.
child open.
child color ← Color blue.
child text ← 'der Kleine'.
child position ← 200@200.

parent add: child.
```

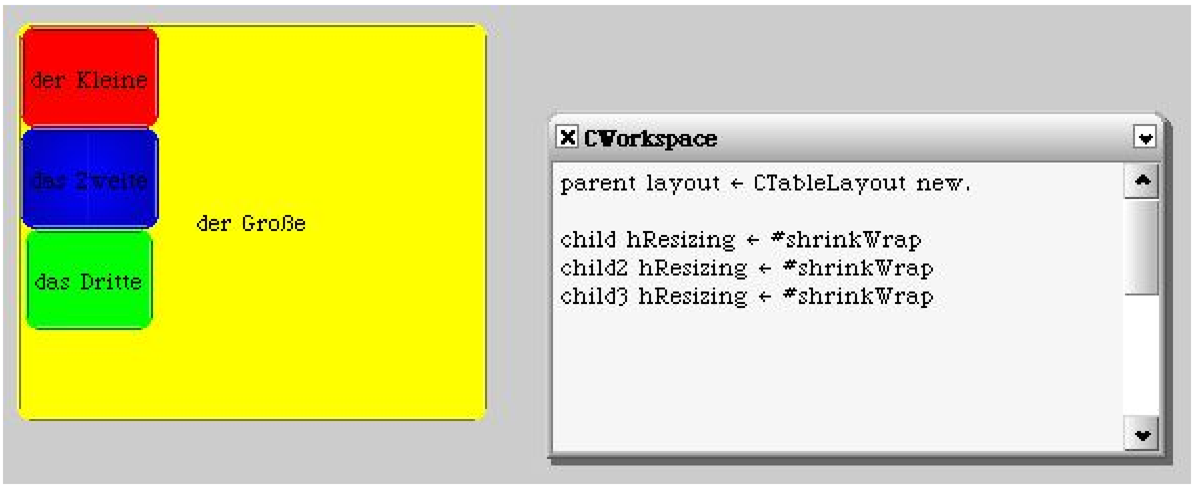
You can assign layouts to the player, to align objects relatively to each other. Layouts can also be assigned by menu or workspace.

TableLayout

To align inner objects on top of each other or side by side, assign the *TableLayout* to the parent object. Using the *TableLayout*, the parameters *hResizing* and *vResizing* are used. The child objects were adapt to the size of the higher ranged one. Depending on their value, all objects have *rigid* size, the higher ranged object is shrunken to the size of the inner objects or the inner objects fill the higher ranged one.







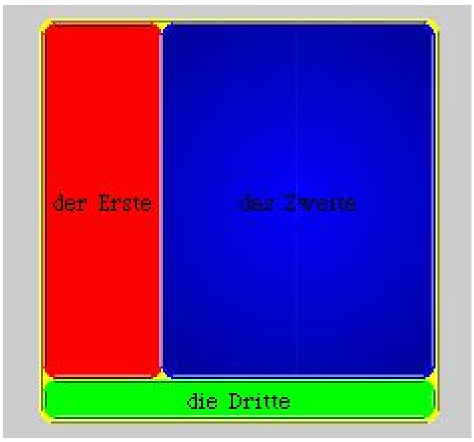
```
parent layout ← CTableLayout new.

child hResizing ← #shrinkWrap
child2 hResizing ← #shrinkWrap
child3 hResizing ← #shrinkWrap
```

ProportionalLayout

To define proportional areas within one object and assign subobjects to, use a *ProportionalLayout*. One example for this is the *browser* object in Squeak. To leave the sizes of all inner object in the same proportion, while the parent changes its size, you have to set the parameters *hResizing* and *vResizing* on *spaceFill*. Positioning of any Playerobjects within the parent object, you have to use *LayoutFrames*.

```
parent _ MyPlayer new.
parent color _ Color yellow.
parent extent _ 200@200.
parent open.

child1 _ MyPlayer new.
child1 color _ Color red.
child1 text _ 'der Erste'.

child2 _ MyPlayer new.
child2 color _ Color blue.
child2 text _ 'das Zweite'.

child3 _ MyPlayer new.
child3 color _ Color green.
child3 text _ 'die Dritte'.

parent layout _ CProportionalLayout new.
parent add: child1.
parent add: child2.
parent add: child3.

child1 hResizing: #spaceFill.
child1 vResizing: #spaceFill.
child1 layoutFrame _ CLayoutFrame
        fractions: (0@0 corner: 0.3@1)
        offsets: (0@0 corner: 0@-20).

child2 hResizing: #spaceFill.
child2 vResizing: #spaceFill.
child2 layoutFrame _ (CLayoutFrame
        fractions: (0.3@0 corner: 1@1)
```

```
            offsets: (Rectangle left: 0 right: 0 top: 0 bottom: -20)).

    child3 hResizing: #spaceFill.
    child3 vResizing: #spaceFill.
    child3 layoutFrame _ (CLayoutFrame
            fractions: (0@1 corner: 1@1)
            offsets: (Rectangle left: 0 right: 0 top: -20 bottom: 0)).
```



Create your own Tweak-Objects

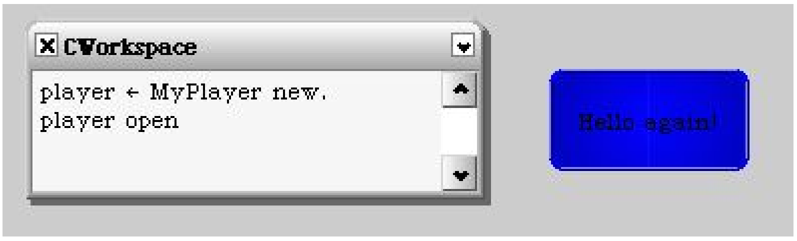
Also, it is possible to define your own Tweak objects out of the Tweak project window. Initializing these objects is possible within the Tweak project window only. Therefor you need to derive your class from class *CPlayer*.

```
    CPlayer subclass: #MyPlayer
            instanceVariableNames: ''
            classVariableNames: ''
            poolDictionaries: ''
            category: 'Tweak-Testsachen'
```

Initializing of the object calls the method initialize, there you can specify the attributes of any instance of your object.

```
    initialize
            super initialize.

            extent _ 100@50.
            color _ Color blue.
            fillStyle _ #radialGradient.
            borderStyle _ #complexFramed.
            roundCorners _ true.
            borderWidth _ 2.
            name _ 'MyPlayer'.
            text _ 'Hello again!'
```

After initializing:



```
player ← MyPlayer new.
player open
```

What is new at Tweak objects compared to other Squeak objects?

Tweak objects do not include instance varables only, they contain fields, too. Changing the instance variables, will not noticed by anyone immeadiately, until the next access on this variable by the referencing objects. Staying the references up-to-date, you need to access the referenced variable regulary. Unlike this, any changes of fields create an event, that is noticed by any interested object.

```
instanceVariableNames: 'foo'
```

After creating a new field, the accessing methods were generated automatically. You can find them in the category *accessing* (Attention: in future versions, these accessing methods will not be visible for the user!).

```
foo
        "Answer the foo of the receiver"
        <bewareOf: #fooChanged>
        ^self propertyValueAt: #foo


foo: aValue
        "Modify the receiver's foo"
        ^self propertyValueAt: #foo put: aValue with: #fooChanged
```

You can choose between *instance variables*, *regular fields* and *virtual fields*, mostly a r*egular field* fits the desired function.
To distinguish between different fields and other variables you can assign colors to the attributes. This way, you can organize the attributes to fields of function that are optically distinguishable.
At the first initialization of any field, there will not be generated any event, however at every change after this it will be. To be informed of any change of the field of interest, the according *annotation* has to be implemented in your method. In our example:

```
        <on: fooChanged>
```

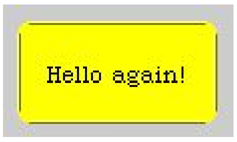
For better readability and distinction of different events, choose a name for the method that anyone can make out the handled event. Within the methods any handling of the event should be implemented. Methods that handle events, should be located in the category *events*.

```
onFooChanged

        <on: fooChanged>
```

```
          Transcript show: 'Foo has changed to: ', self foo          asString; cr.
```

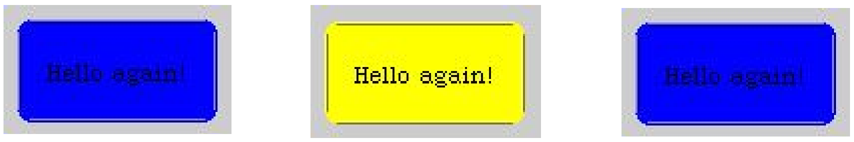
Another event: handle any *mouseDown* event:

```
onMouseDown

          <on: mouseDown>
          color _ Color yellow.
```



At our example, the *CPlayer* gets a new color after the user has clicked on it. But, after that happens, the old color is lost. To reset the color after the *mouseUp* event reaches, you have to store the old color temporarily until the *mouseDown* event is finished. Therefor Tweak offers a simplified mechanism:

```
onMouseDown

          | oldColor |
          <on: mouseDown>

          oldColor _ color.
          color _ Color yellow.

          self waitUntil: #mouseUp.
          color _ oldColor.
```

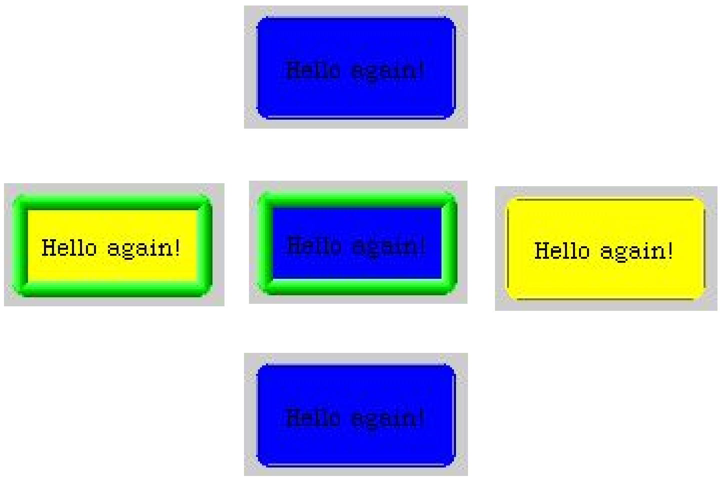


The execution of the *onMouseDown* method is stopped using the *waitUntil:* until any *mouseUp* event is reached.[1]

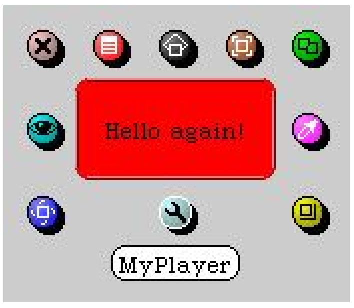
1 Internally, any event is handled by its own thread. Threads are not interrupted, except they get an explicite *wait*, which suspends the thread until any special event is reached. Furthermore, the same event is not triggered twice, while the according thread is not finished.

A simple example for this mechanism are the independent events *mouseEnter* and *mouseDown*. While staying clicked at the object (*mouseDown*), the object is already yellow colored, you can leave the object bounds with the mousecursor and cause another event, i.e. *mouseEnter* or *mouseLeave*.

```
onMouseEnter

        | oldColor oldWidth |
        <on: mouseEnter>

        oldWidth _ borderWidth.
        oldColor _ borderColor.
        borderWidth _ 8.
        borderColor _ Color green.

        self waitUntil: #mouseLeave.
        borderWidth _ oldWidth.
        borderColor _ oldColor.
```



Any changes of the fields are noticable using the viewer.



Special things of the mouse events

Mouse events are global, that means the primary position of the mouse is relative to the window (squeak window). If you need the local position of the mousecursor you have to call the method *globalToLocal* or *cursorPoint*, that returns the local point directly.

Draw your own Tweak-Object by yourself

Already it is possible to draw objects by yourself instead doing it automatically by the system. Therefor you have to set the attribute *userDraw true* within the *initialize* method.

```
self userDraw: true
```

Furthermore, you have to implement the method *drawOn: in:* . If you do not implement this method or call this method of the superclass only, beyond, drawing is done automatically.
To decide which things are able to draw, which conditions have to be fulfilled, is going too far at this moment. Have a look at the class *CTransformCanvas*.