

Maui

by

Chris Muller

Maui	1
Introduction	4
Guiding Principles	4
Core Widgets	4
Views	5
Architecture	5
Selecting Views	6
System-Views	6
The #panel view	6
The #defaultListEntry view	7
The #word view	7
Object Delineation	8
User-Interface Conventions	10
The Context Menus	10
Menu Layout	11
Hot Keys	11
Panel Management	11
Clipboard.....	12
Labeling.....	12
Messages	12
InvocationStrategies	13
Invoking Unary Messages	13
Invoking Binary and Keyword Messages.....	13
Supplying ParameterHolders via Drag-and-Drop	14
Supplying ParameterHolders via typing an expression	15
Parameter-Holder Evaluators	15
Smart-Sizing ParameterHolders	16
Supplying ParameterHolders via "Pick Lists"	17
Combo-box Pick Lists	17
List-box Pick Lists.....	17
Checkbox Pick Lists	18
Abitrary Pick Lists.....	19
MauiPage.....	20
Architecture	20
Implications for Maui.....	22
Construction of a MauiPage	22
Other Specialty Widgets.....	23
MauiCollectionMorph	23
MaContextualSearch	24
MaClientProcessMorph	25
Color Pickers	25
Date Picker	25
File Manager.....	26
Preferences	27
Object-Message Composition.....	29
Obtaining an Object to Design	29
About PlotMorph.....	31

Deleting an object.....	31
Specifying Message Settings	32
Changing The Label of an object.....	34
Designing the UI.....	34
A Quick-and-Dirty Composition	34
Sub-panel Composition.....	35
Create the sub-panels	35
Changing the resultView of a MessageMorph widget.....	37
About InvocationStrategy's	37
Making a Getter/Setter.....	39
Specifying an Object "Picker"	40
The MauiBehaviorFinder	42
Completely Customized Looks	44
Saving Custom Panel Configurations.....	45
Panels are Designed, now what?	48
Assembling Panels	48
Exercise: Creating the Series Panel.....	49
Customizing the list-entries of a MauiCollectionMorph	50
The Default Behavior	50
Making a Custom List-Entry	51
Adding Messages to the context menu	52
Roadmap	54

Introduction

Maui is a tool for rapid UI creation based on object--message composition, a.k.a. "naked objects". Compositions can be saved as a named prototype, used in super-compositions, and grouped into a Family of related panel prototypes all serving one domain.

A Smalltalk developer might describe Maui as a "behavioral inspector."

Maui includes a number of light satellite frameworks that supply various application services like documents, object-search, background process management with progress monitoring. Maui allows utilitarian applications to be synthesized quickly, modalelessly, without the need to write any user-interface code.

Guiding Principles

Maui's goal is to present user-interfaces with high functional-density, efficient access, and low clutter. Completely modaleless, Maui tries to be smart about sizing and placement, since user move and size actions drain time away from the domain itself.

Core Widgets

Maui's three core widgets are:

- ***MauiDomainMorph*** - a rectangular Morph representing a single object. Rendered as a compact composition of messages. Multiple views of the same object are easily produced and used.
- ***MauiMessageMorph*** - one of several sub-rectangles, embedded into the MauiDomainMorph, ready to send its *MauiMessage* object as a message to the underlying receiver of the MauiDomainMorph.
- ***MauiParameterHolderMorph*** - Embedded into the MauiMessageMorpha, parameter-holders are rectangles where other MauiDomainMorpha are dragged as arguments of that particular MauiMessageMorph.

A MauiDomainMorph can be created, rendered and attached to the hand by sending #maui to any object.

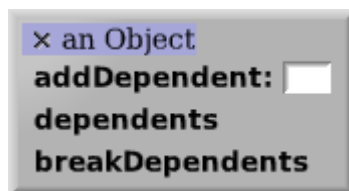
```
anObject maui
```

Example:

From a workspace, this code

```
"demo"  
Object new maui  
  addMessageNamed: #addDependent: ;  
  addMessageNamed: #dependents ;  
  addMessageNamed: #breakDependents ;  
  yourself
```

produces this panel, attached to the mouse pointer (a.k.a., the "hand").



The main rectangle is the DomainMorph.

It has a label, "an Object", then three MessageMorphs.

The first of the MessageMorphs has a ParameterHolderMorph.

Views

Architecture

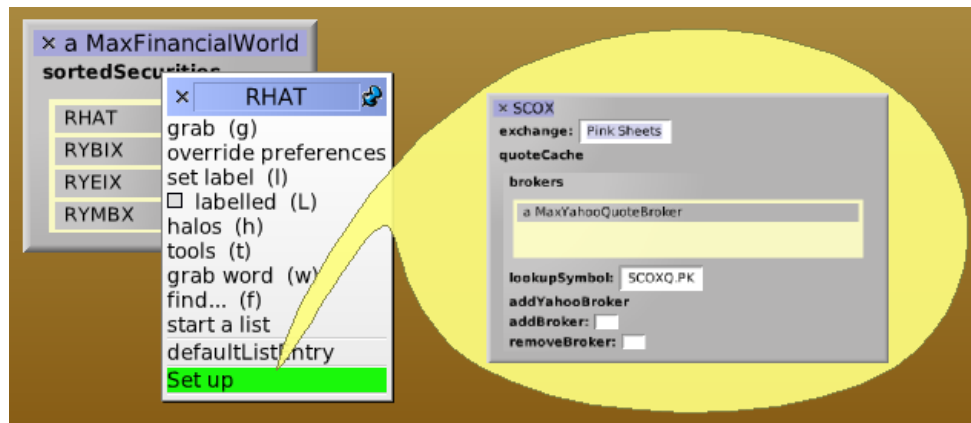
Every "panel" rendered by Maui is an instance of a MauiDomainMorph. Each instance can define its own set of messages to render, as well as its own layout and other aspects. A particular configuration of a particular class of object is referred to as a *view*.

The "configuration" of a particular view is actually represented by an exact clone of the original view (a *prototype*), stored with other panels related to the same overall domain, in a MauiFamily. Prototype clones can then be requested by name and class (MauiFamily>>#viewNamed:for:), so any kind-of that object can be represented by that panel, including subclass instances. Subclasses may override the more-generic super panels in order to offer their extended subclass behavior.

Once the MauiFamily's copy of any of its named prototypical instances is obtained, it can then be injected with a suitable domain object using the #object: setter. All of the messages embedded into that view will reflect the answers for that object.

Selecting Views

Customized views appear on the usage context menu, just below all of the standard options.



The small bitmap in the balloon was captured when the panel was saved as a reminder of how it looked.

System-Views

Besides user-defined views composed by UI designers (see Object--Message Composition), Maui also has pre-defined views which are code-generated. The names of these system views are represented by selector Symbols rather than just simple Strings. Those symbols are performed on (sub)instances of MauiDomainMorph, which programmatically add a set of related, useful messages and sets up various user-interface conveniences.

The #panel view

#panel is the standard view when no other view is specified. There are some basic kinds of objects supplied with the rich Squeak 3.9 environment that users shouldn't have to build themselves. For example, Strings, Numbers, Collections, or even FileDirectory's. The following workspace opens a default #panel view for the current month:

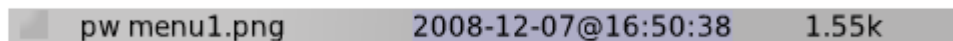
```
Date today asMonth maui
```

Which presents:



The #defaultListEntry view

Another system-view is called #defaultListEntry. This is a quick-and-easy way to have meaningful information when a particular class of object is presented in a Collection (via MauiCollectionMorph). This is one of the instances of DirectoryEntry from the FileDirectory above:



The embedded MauiMessageMorphs are defined by #mauiDefaultColumns.

```
DirectoryEntry>>mauiDefaultColumns
^#( name modificationDateAndTime maFileSizeString )
```

This is used by the hard-coded #defaultListEntry method to add messages with those names to a wide-and-short panel laid out horizontally. Each message added specifies a #resultView of #word, which is one of the most useful views due to its compactness.

Note: The object may be "torn off" these list entries for dragging into the argument of some keyword message.

The #word view

The #word view, presents a compact rendering of an object with only the objects #mauiName (default implementation in Object uses its #printString).



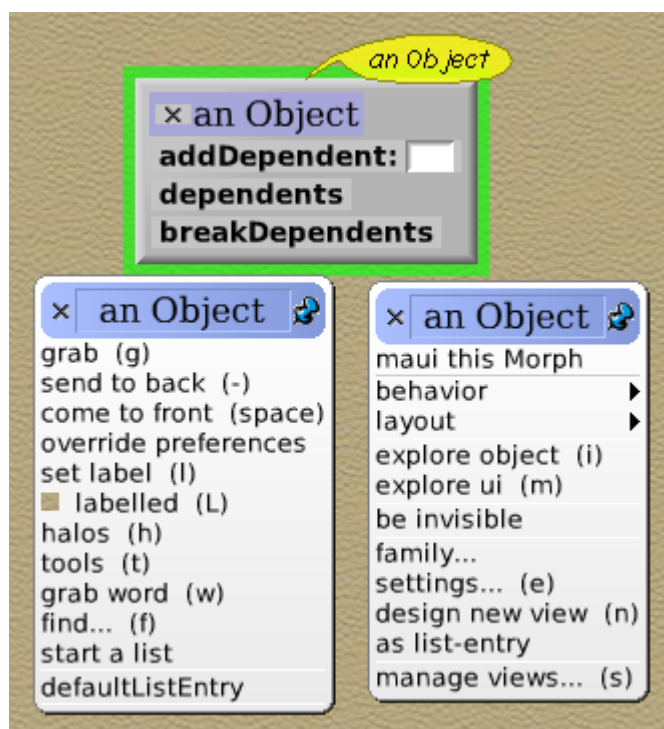
In #word view, the object has a fillStyle specified by its individual or inherited #**wordFill** Preference (see "Preferences"). This helps users remember it is really the entire object, not just a String.

Object Delineation

While graced with the focus of the hand, an object is highlighted, as well as all identical occurrences of that object visible on the desktop. If the Connectors project is loaded from SqueakMap, translucent lines will be drawn to the other instances.

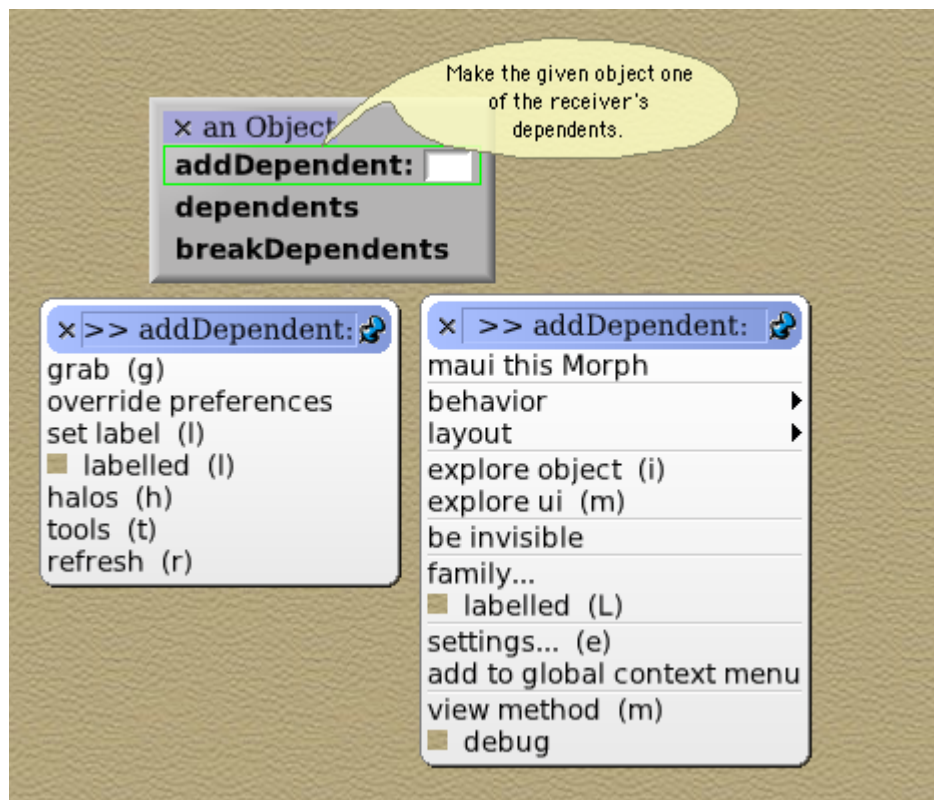
Each of the other widget types also delineate themselves by highlighting their border while graced with the focus of the hand. Additionally, each supplies its own balloon-help, context menu and hot-key mapping.

This shows our example Object, delineated, its balloon-help description, its standard right-click "operational" context menu (right-click) and its "tools" menu (Shift + right-click). The highlighted object responds to a number of hot-key commands, not all of which are available on the menus.

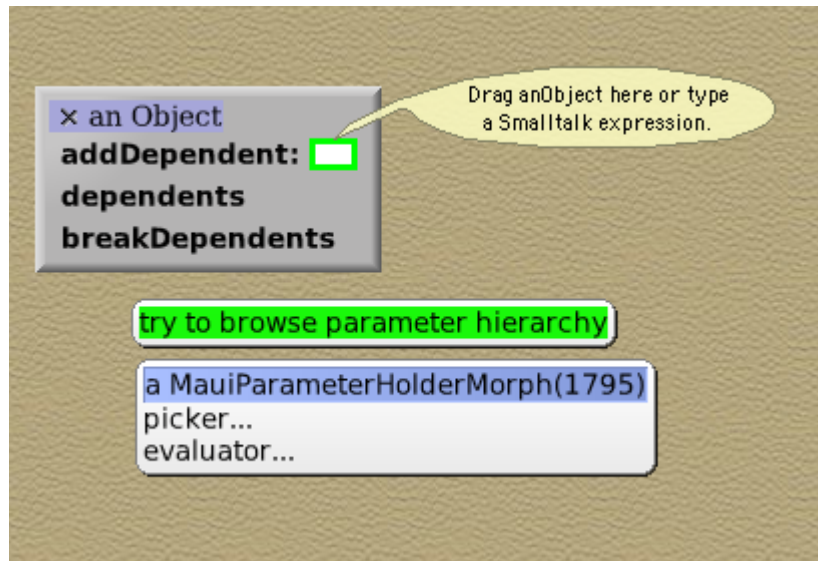


To find out what each menu item does, hover over it with the hand, a pop-up balloon displaying the invoked methods first comment is displayed.

This shows our example object with the mouse over its #addDependent: message. The balloon-help shows the first comment of the method. It has its own operational and tools menus, and its own hot-keys.



This shows the parameter-holder is also its own object with its own functionality. The balloon-help in this case informs the user what type-suggesting parameter name the developer used for the first argument, "anObject". This is the kind of object that needs to be dragged to the parameter-holder to properly invoke the #addDependent: message (but since any object can be a dependent, it is actually the most generic type possible).



The result is a high functional-density with efficient access. Compared to conventional interfaces, Maui interfaces require less physical "work" from humans in terms of mousing distance, button presses, and clicks. Not every action is bottlenecked through a "left-click" of the mouse.

User-Interface Conventions

A desktop with just 10 open Maui panels can easily result in more than 100 separate event-monitoring regions on the screen, each with at least 10 independent "commands" that could be invoked via mouse-buttons, hot-keys, or menu selections. These 1000 separate functions can all be reached with the same simple lifecycle:

1. Look.
2. Point.
3. If necessary, press a button.
4. If necessary, select a desired menu command.

Many commands are accessible via hot-keys, which can spare a fourth gesture in most cases. In some cases it is not even necessary to press a button. Maui allows "mouse-over" message invocation is easily specified (see *Object-Message Composition*).

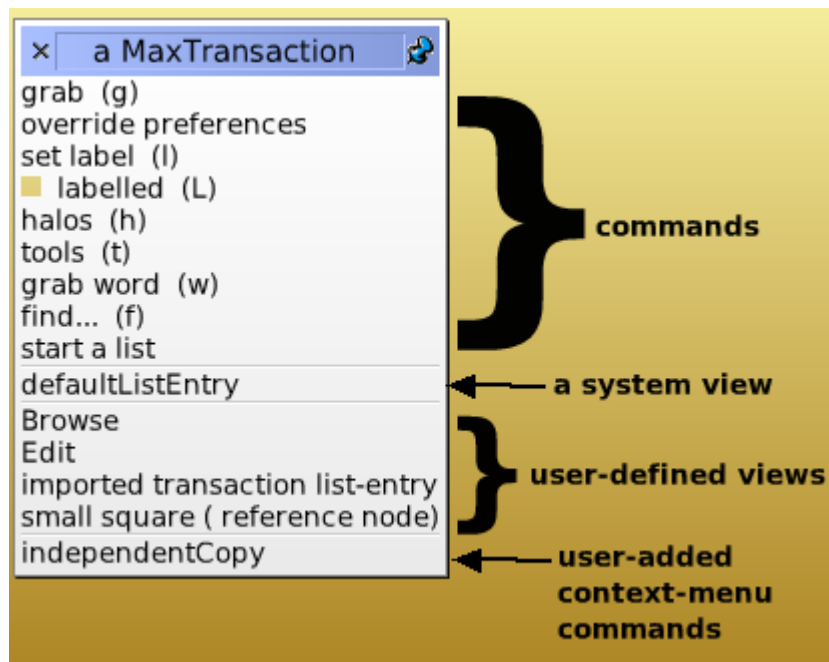
The Context Menus

Every Maui object has two context menus. One with items meant to **use** the object in an application is invoked merely by right-clicking, or pressing the Escape key. The

other has meta items meant to further "design" the panel, and is presented from the first menu by selecting the "tools" option, or by holding down the Shift key while right-clicking on the object. Another way to reach the tools menu, point at the object and press the lowercase 'T' command key.

Menu Layout

The standard context menu is presented by right-clicking on an object. For DomainMorphs, the context menu is arranged in a group of sections:



The letters in parenthesis in the menus are the hot-key equivalent for that function.

Hot Keys

It should be noted it is not necessary to press Alt or Control when using hot-keys, leaving another hand more free to point.

Maui's hot-keys are completely customizable (see "Preferences"), and using them can result in faster operation of a user-interface than always invoking the menus.

Panel Management

There is one MauiWorld for each Morphic World. To obtain the MauiWorld for the current World:

MauiWorld current

MauiWorld provides methods related to the panels visible on the desktop. It would be nice to make a UI for it, it'd be an automatic "window finder."

Clipboard

Objects cannot be put onto the system clipboard. However, a copy of any object can be pasted within Maui at any time. To do this:

1. Point at the object to be copied.
2. Press the lowercase 'V' key (paste key), a copy of the receiver view is attached to the hand.

Every time the lowercase 'X' command key is used for removing an object, that object is replaces the contents of the current MauiWorld's #clipboard.

Labeling

Besides endowing panels with a familiar "windowish" look, the purpose of labeling is to provide identifying information. Maui presents the #mauiName as the default label, but allows labels to be overridden on an object-by-object basis, which can be useful for context-specific identification or "investigations."

To change the label of any panel or message, point to it and press the lowercase "L" key on the keyboard (or select the "set label" option from the "usage" context menu). To completely hide the label of a panel or message, press the uppercase "L" on the keyboard. The standard hot-keys chosen for Maui sometimes have a related "Shift" alternate that usually invokes a design command.

Messages

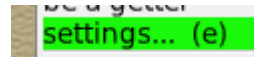
This section describes how and when messages are invoked, and what happens when they are. By default, messages which are not customized by a designer are handled conservatively as follows:

unary - invoked by left-clicking. The answer is provided in #word view by default, embedded in the MauiMessageMorph.

binary - same as keyword

keyword - invoked by dragging MauiDomainMorph objects to all ParameterHolders of a MessageMorph. By default, the message is invoked automatically as soon as the last ParameterHolder is filled, and all objects dropped into the ParameterHolders are then removed.

Designers can customize this behavior by pointing to the message and then pressing the lowercase 'E' key (for "Edit") on the keyboard, or by selecting the



option from the **tools** menu. This is explained more fully in "Object--Message Composition".

InvocationStrategies

The MauiMessageMorphs embedded into domain objects are invoked according to their *InvocationStrategy*. An *InvocationStrategy* is an instance of *MauiInvocationStrategy* which specifies the following:

- What event should cause the message to invoke.
 - *clicked-on* - for messages that perform some kind of action.
 - *moused-over* - the message is invoked every time it receives focus from the hand. Useful for tucking-away information that still wants to be checked periodically.
 - *auto* - the message is invoked when the object is initially opened, then according to its *#repeat* and *#repeatInterval* parameters.
- Where should the result of the message be placed, a.k.a., *#resultHolderTarget*:. The possible values are, 'balloon', 'hand', 'local', 'world' or 'none'.
- What view should the result object assume.
- Which view should be the "tear-off" view, a.k.a., the *#producedView*. Setting this to nil causes the result object to be picked up by the hand when left-clicked.

Invoking Unary Messages

Unary messages don't require any arguments, therefore they require clicked-on to invoke, by default. It can be changed to invoke only when the underlying object *#changed* or even on a repeating basis.

Invoking Binary and Keyword Messages

In Smalltalk, a *keyword message* is a message which takes arguments. To invoke the message, objects must be supplied to each argument of the MauiMessageMorph. This can be done in the following ways:

- dragging any MauiDomainMorph into the parameter-holder and dropping it there.
- typing an expression directly into the parameter-holder.
- using, if provided by the designer, the "picker", which presents a series of common objects to pick from.

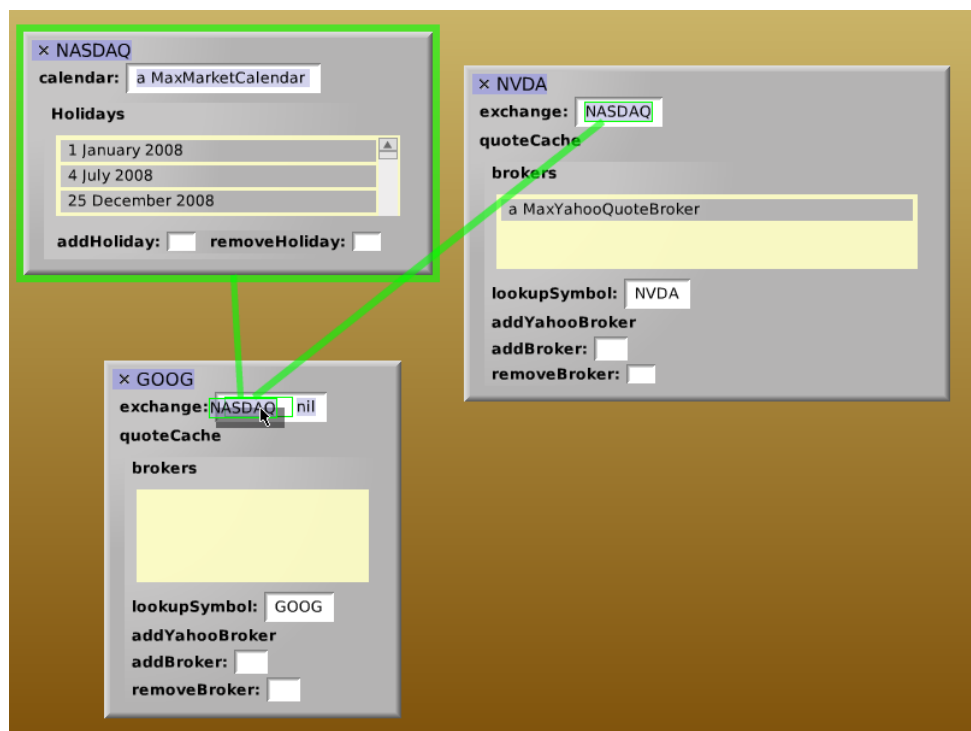
We will now look at each of these in detail.

Supplying ParameterHolders via Drag-and-Drop

Many user-interfaces establish complex domain references by presenting a pop-up list of complex objects from which they can select. Maui can do this too (see "Optional Parameter Settings", below), but also allows any object visible anywhere on the desktop, even embedded in another Morph, to be dragged in directly as an argument to the message.

In this Maui-based financial application, the user is setting up the object representing a particular financial security known as "GOOG". Shares of GOOG are traded on a particular FinancialExchange named "NASDAQ". In this screen shot, the NASDAQ object is visible in three separate view instances, each delineated by green outlines. The one in the upper-left is a user-defined view they named 'Calendar'. The other two are the standard #word view previously described.

Since the user knows there is only one FinancialExchange named "NASDAQ", and is the one referenced by the NVDA security, there was no need to go elsewhere looking for it, NVDA is also traded on the NASDAQ, the object is right there.



This also illustrates the "tear-off" (e.g., #producedView:) function of any MauiDomainMorph. The upper-right NASDAQ object has its #producedView set to #word, which tells the object, when left-clicked, rather than pick it up, pick up a UI

copy (but exact same domain object) in the same #word view.

Also illustrated, are the translucent lines used to connect all occurrences of that identical object on the screen. These are only available if the Connectors package is loaded (installable from directly within the image via the "SqueakMap Package Loader").

Supplying ParameterHolders via typing an expression

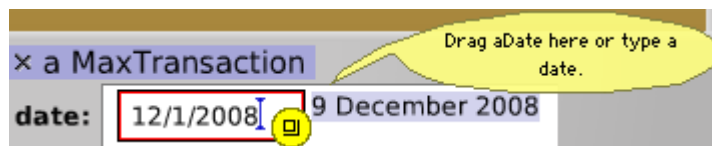
Another way to supply objects to message parameter-holders is by typing directly into the parameter-holder morphs. To do this:

1. Point at the parameter-holder. It is delineated.
2. Note the balloon help, which hints at what kind of MauiStringEvaluator is used.
3. Begin typing. It is not necessary to left-click in the parameter-holder, although doing so won't hurt anything.

Parameter-Holder Evaluators

What is typed into parameter-holders will be interpreted according to how the parameter-holder was set up by the designer. Maui includes three interpreters, one for standard Strings, one for Dates and one that evaluates as a Smalltalk-expression. Applications designers may add their own interpreter subclasses.

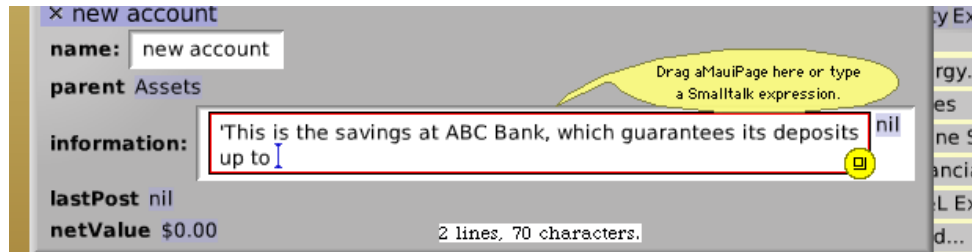
- This balloon indicates to "type a date", which means keyboard entry will be interpreted by a MauiDateEvaluator.



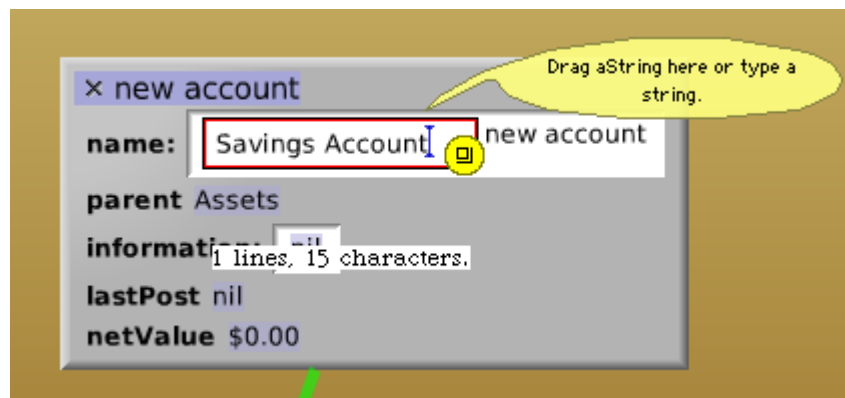
Looking at MauiDateEvaluator class, we see the method and how it converts the actual inputted String into an appropriate object:

```
evaluate: aString for: anObject  
  ^ Date fromString: aString
```

- This balloon indicates to "type a Smalltalk expression," meaning anything typed in the box will be evaluated as a Smalltalk expression in the context of the receiver. That is why, to drop a plain String here, the user must type the single-quote delimiters.



- It also indicated the user can drag "aMauiPage" there. A MauiPage is Maui's Text domain object, supported by an included mini-wordprocessor widget (see "MauiPage" later).
- This balloon instructs to "type a string", indicating it will treat what is entered as a literal and drop it as a String. Therefore it is not necessary to type the single-quote delimiters.



The default is a MauiSmalltalkEvaluator, which means the typed expression is compiled and executed in the context of the receiver object. The result is then dropped into that parameter-holder.

Smart-Sizing ParameterHolders

In keeping with its generic, compact nature, typing into parameter-holders causes them to expand as you type; first horizontally then, after reaching a reasonable proportion of the screen-width, wrap to a new line. Continued typing will maintain a 3x4 width:height ratio until, when the size expands beyond a reasonable proportion of the total screen-height, will add a scroll-bar.

In this way, whether a field is short-entry, multi-line long-entry, or anything in-between, there is only one widget to use. The design goal was to produce a widget that always adjusts its size to be as small as it reasonably can.

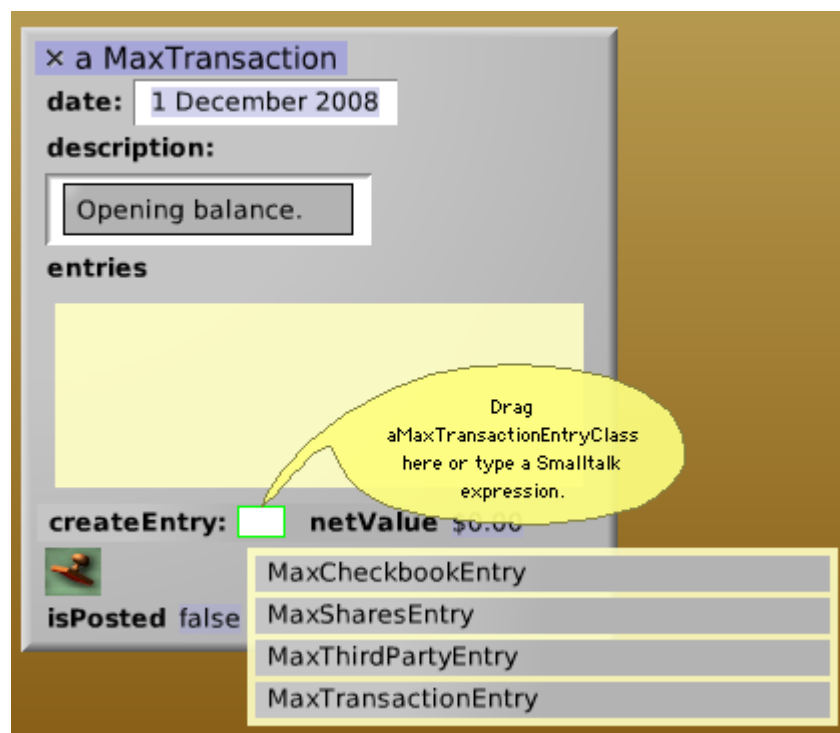
The yellow-circle may be dragged to override automatic sizing at any time.

Supplying ParameterHolders via "Pick Lists"

Pick lists are very common in standard user-interfaces, Maui provides a simple "pick-list" function. Although **any MauiDomainMorph** can act as a pick list, there are three familiar forms provided for easy implementation of the most-common standard list-widgets: *combobox*, *listbox*, and *checkbox*.

Combo-box Pick Lists

Combo-box Pick Lists are obtained by left-clicking on the parameter-holder. A Collection of available options is displayed just below the parameter-holder, left-clicking any of them will drop it into that parameter-holder immediately.

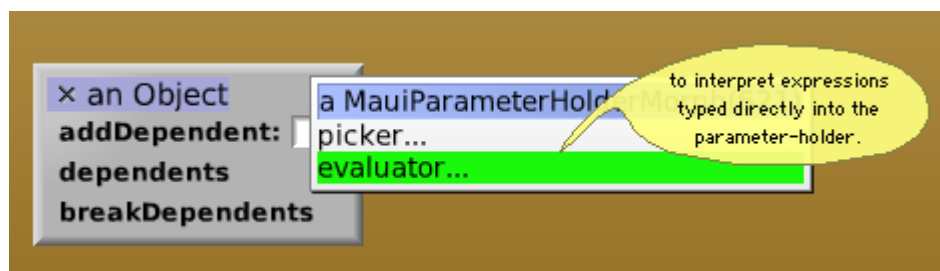


Maui makes it easy to specify your own pick-lists, either fixed or program-generated. This will be described under "Object-Message Composition."

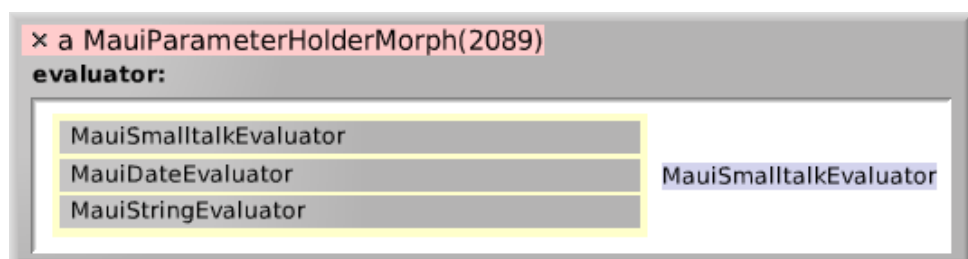
List-box Pick Lists

List-box pick lists are identical to combo-box, except they display all the time, taking up more screen space but not requiring a left-click to initially see the list. To demonstrate the List-box pick list, we will sneak peek at a upcoming section and see how a particular Evaluator is specified for a particular ParameterHolder.

Hover over any parameter-holder, then Shift + Right-click. The following menu appears:



After selecting that option, the three default MauiStringEvaluators are presented in a **list-box** format:



The current selection is centered on the right. Left-clicking any of the other selections will immediately drop that object into the #evaluator: message, instructing the other ParameterHolder how to interpret strings.

Which brings up another note, Maui opens panels on its own panels or widgets to customize some of their functions. These "utility" panels, opened on a kind of MauiDomainMorph, display their label bar in a different color (customizable, see "Preferences"), to indicate they are "designer" panels operating only on a UI panel, not on any domain object.

Checkbox Pick Lists

Checkboxes are a very common way to indicate Boolean value. But Maui genericizes it by making it about a Collection of parameters that are "cycled" each time the user left-clicks on them or the parameter-holder. There can be more than two. A "checkbox," then is simply a collection of the object **true** and **false**.

In this screenshot, a Maui panel is being used to customize a PlotMorph.

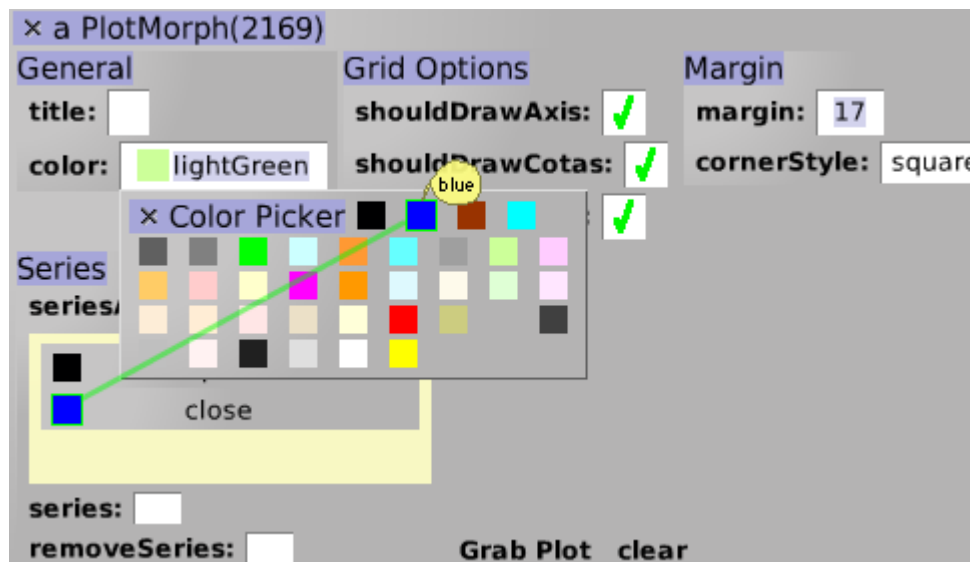


Remember that any object, including the **true** and the **false**, can have multiple views. This designer used Squeak's built-in Paint tool to create his own 'iconic' representations of true and false, represented as a green checkmark and a red X, respectively.

Also interesting note about this panel illustrates the generic nature of Maui's "checkbox"; that the `#cornerStyle` message also uses a `CycleParametersStrategy` to cycle between the Symbols `#square` and `#rounded`, merely by left-clicking them or the parameter-holder.

Arbitrary Pick Lists

Maui's pick-lists are pretty flexible. Finally, you can use virtually any `MauiDomainMorph` as the "picker". In this example, a Maui panel is being used to customize a `PlotMorph`. The user is about to set the (background) `#color:` of the `PlotMorph`.



The user has hovered over the Color blue, that is why the translucent green line is drawn to every other occurrence on the screen; relevant or not, its the same object.

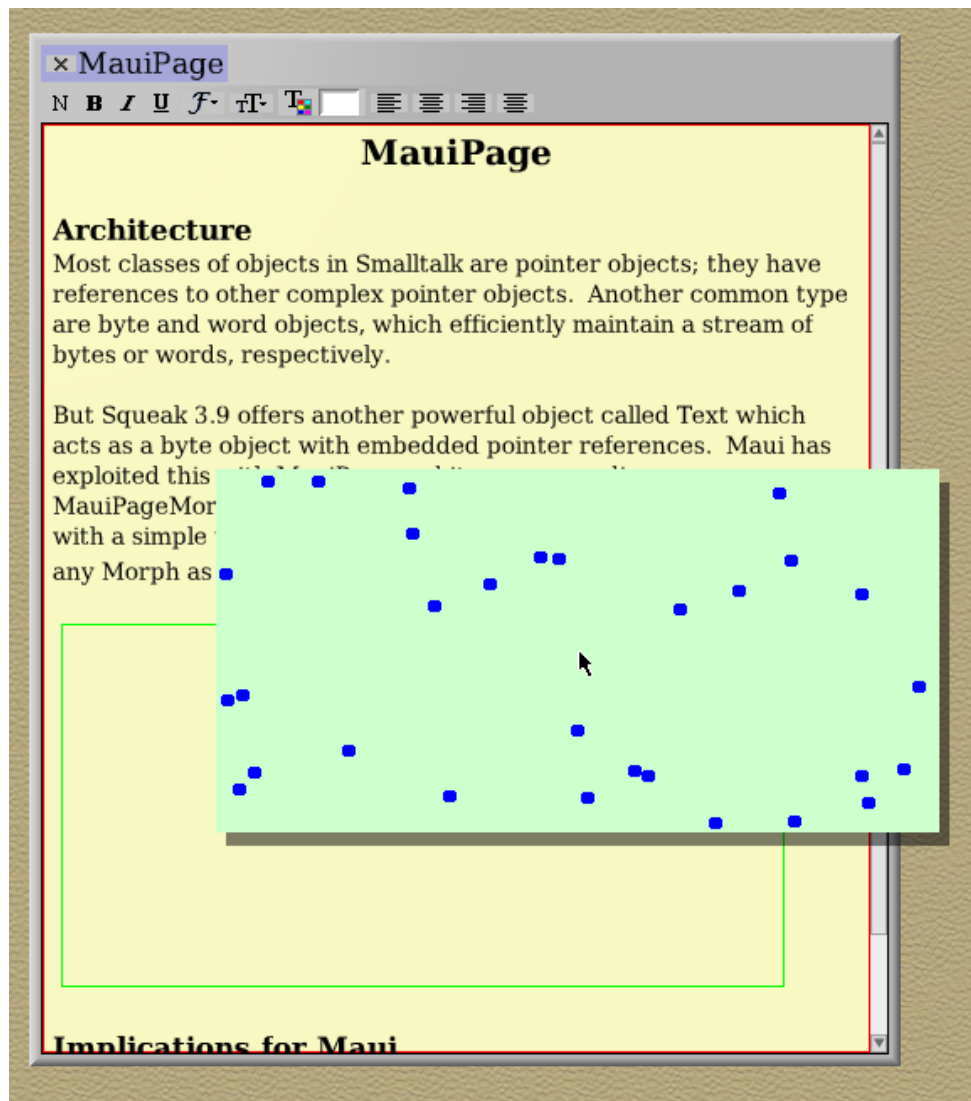
Another use of the generic picker would be to use Maui's own MauiFileDirectoryMorph to act as a "file picker".

MauiPage

Architecture

Most classes of objects in Smalltalk are *pointer* objects; they have references to other complex pointer objects. Another common type are *byte* and *word* objects, which efficiently maintain a stream of bytes or words, respectively.

But Squeak 3.9 offers another powerful object called Text which acts as a byte object with embedded pointer references. Maui has exploited this with MauiPage and its corresponding MauiPageMorph. This specialty widget is like a mini-wordprocessor, with a simple toolbar for customizing text, and the ability to embed any Morph as a "character" in the text:



A MauiPage with the user dragging in the famous "bouncing atoms" Morphic demo.

As with everywhere else in Maui, the green outline indicates exactly where the morph will be dropped.

A deepCopy of the morph is what is actually dropped.

Once embedded, it flows and wraps just like a big character, and can easily be deleted with the backspace key!

Every time the page is "saved" (Command+s), it saves a copy of itself.

It's interesting to note that the underlying widget used for the MauiPage is the same one used for ParameterHolder morphs, a testament to its flexibility!

Implications for Maui

"Any Morph" is nice for Squeak, but the embedding of other MauiDomainMorphs presents a number of additional opportunities for leveraging the capabilities of Maui:

- **Domain validation:** MauiPages can be signaled from domain applications as "error messages" with embedded objects that need correction.
- **Training / job-aids:** MauiPages can be used to write documentatoin with embedded live, working objects.
- **End-user empowering:** MauiPage fields can allow users establish inter-object relationships without needing to rely on program functionality. This can eliminate the need for entire chunks of program code.

Construction of a MauiPage

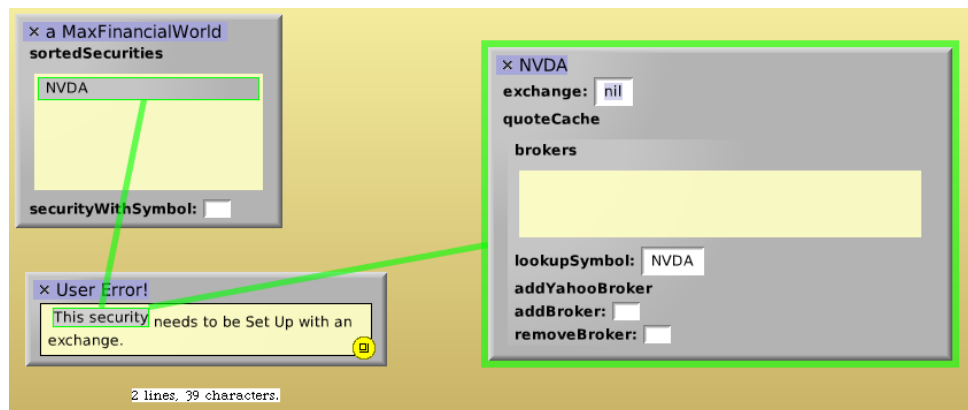
Maui pages can be constructed programmatically with a Streaming api, or via the MauiPageMorph specialty widget. To use the specialty widget, execute the following in a workspace:

```
MauiPage new maui
```

Applications wishing to leverage Maui may:

```
MauiUserError signalPage: myMauiPageWithErrorMessage
```

which will cause Maui to present a useful message to the end-user with objects embedded that need fixed. For example, in this financial application, the user tried to access the trading #calendar for the NVDA security. But the calendar is based on what exchange the security is traded on, and the exchange for this security has not yet been set up.



The user right-clicks on This security and selects the 'Set Up' view from the menu, which presents the view on the right to set the #exchange:. If the #producedView: is set, the user would be able to simply "tear-off" the appropriate panel for setting up the exchange.

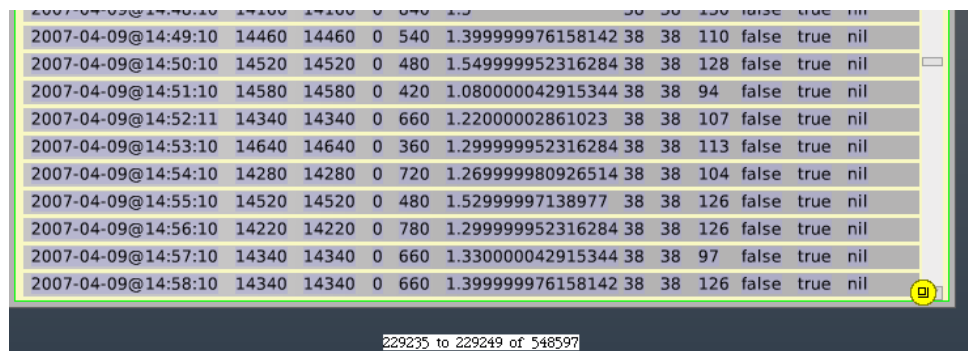
The application was able to leverage Maui's labeling function to allow the error-message to flow with proper grammar. "This security" is just the label for the NVDA object instance is embedded in #word view, but with its label set to "This security".

Other Specialty Widgets

The MauiPageMorph is one of Maui's most useful "specialty" widgets. There are a number of others included with the base package.

MauiCollectionMorph

We have already seen MauiCollectionMorph. It's most notable feature is that it never enumerates all elements to do any of its work. Unlike other list widgets, therefore, a MauiCollectionMorph is not limited in how large a collection it can display.



2007-04-09@14:49:10	14460	14460	0	540	1.3999999976158142	38	38	110	false	true	nil
2007-04-09@14:50:10	14520	14520	0	480	1.5499999952316284	38	38	128	false	true	nil
2007-04-09@14:51:10	14580	14580	0	420	1.0800000042915344	38	38	94	false	true	nil
2007-04-09@14:52:11	14340	14340	0	660	1.220000002861023	38	38	107	false	true	nil
2007-04-09@14:53:10	14640	14640	0	360	1.2999999952316284	38	38	113	false	true	nil
2007-04-09@14:54:10	14280	14280	0	720	1.2699999980926514	38	38	104	false	true	nil
2007-04-09@14:55:10	14520	14520	0	480	1.529999997138977	38	38	126	false	true	nil
2007-04-09@14:56:10	14220	14220	0	780	1.2999999952316284	38	38	126	false	true	nil
2007-04-09@14:57:10	14340	14340	0	660	1.3300000042915344	38	38	97	false	true	nil
2007-04-09@14:58:10	14340	14340	0	660	1.3999999976158142	38	38	126	false	true	nil

The object delineated in green is a MauiCollectionMorph whose underlying domain object is a MagmaCollectionReader with more than one-half million elements in it. Again, any object which can answer #size and #at: is supported. The initial population, scrolling and paging all operate with sub-second response time, the same as if the list had only 100 elements.

It's ironic for Maui to support large lists this well, since they seem to go against the core principle of Maui, that the computer should cater to the user. Large-scrolling lists require the user to do a lot of scrolling and visual-scanning with their eyes to locate elements. Computers should be doing the locating, not people, which is why

Maui provides a generic "object search" tool so the computer can do the scanning.

A large scrolling list can convey a lot of information efficiently, notably its size and contents type are quickly ascertained. A sample of the elements is immediately available without having to send the #anyOne message. Finally, a quick scroll around can provide a "feel" for the contents of the data. Finally, many users are familiar and comfortable with large scrolling lists, so it made sense to design it so.

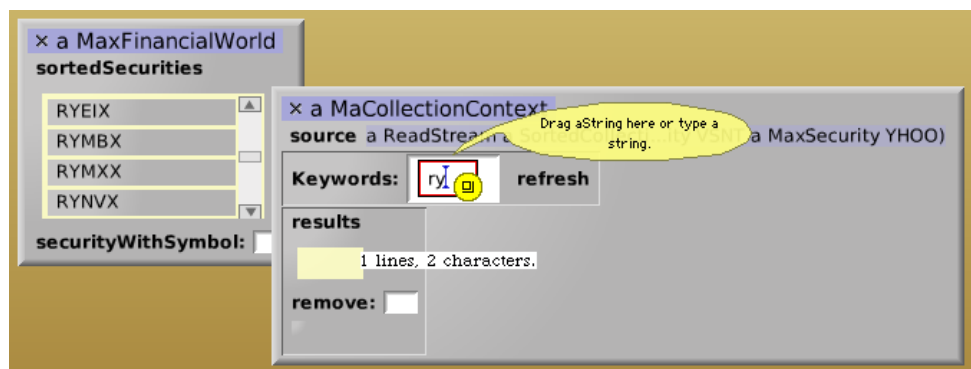
MaContextualSearch

Maui includes a set of light frameworks to support common application functions. One such function is the ability to "find" objects matching one or more keywords. Typically applications provide this themselves, but MaContextualSearch has the ability to provide at least a basic "find" function without having to write a single line of application code.

The framework itself is fit for separate document all its own, but we will show how it is invoked from within Maui:

1. Point at the object to search.
2. Press the lowercase 'F' key.

Maui opens a basic "find" panel. One or more keywords can be typed into the "Keywords" field. Press Enter and the search commences.



Characteristics of Maui searching are:

- The scan executes in the background.
- Objects have multiple keywords, at least one must match loosely match.
- Objects implement #maContextKeywordsDo: and value aBlock for each of their keywords.

- The results discovered "so far" are displayed in the results list for further inspection by the user even while the background search proceeds.
- The order of the results is prioritized according to the degree of keyword match, as follows:
 - Exact, whole matches.
 - Case-insensitive, whole matches.
 - Left-side matches (case-insensitive).
 - Sub-string matches (case-insensitive).
- The user gets to maintain the "results" collection, manually paring it down or subsequent adding searches.
- The results, themselves, may be searched.
- SearchContexts may be aggregated into larger super-contexts. For example, one may wish to search about the Squeak compiler. Aggregating several contexts could allow the local image, swiki, mailing list archive all composed into one search-context, with results each on their own tab.
- Progress is monitored via a MaClientProcess, providing detailed progress and estimated time remaining.

This function is lightweight enough to be used just to avoid scrolling too much. Just point at any MauiCollectionMorph, it is highlighted. Press lowercase 'F', type in keyword, press Enter. Objects are presented. Sometimes the direct route and let the computer do the work is easier.

MaClientProcessMorph

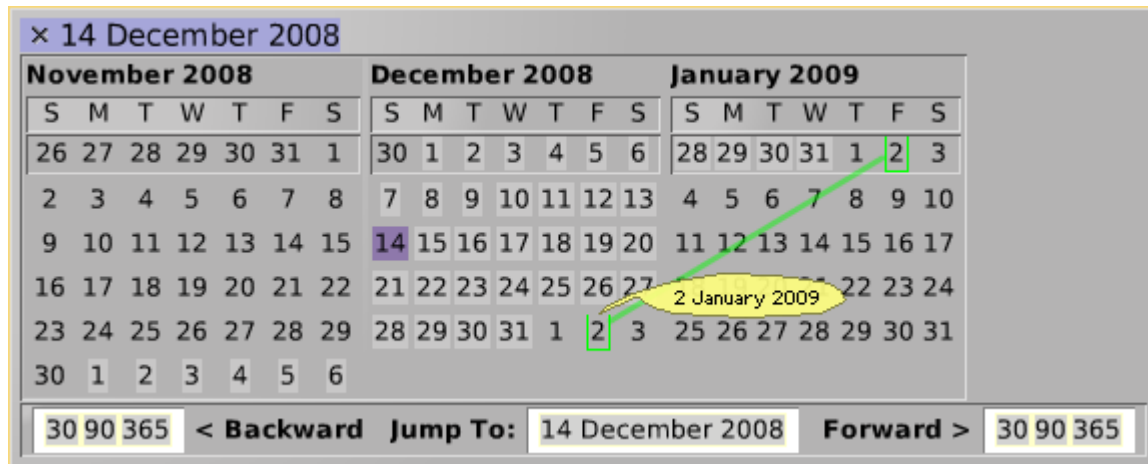
As mentioned in the previous section, there is a generic background process monitor. The framework itself is fit for discussion all its own, but its workings can be easily explored within Maui.

Color Pickers

We have seen the namedColors picker. There are other panels though, for fine-tuning color selections. Right click on a Maui'd Color instance to see the available system-views.

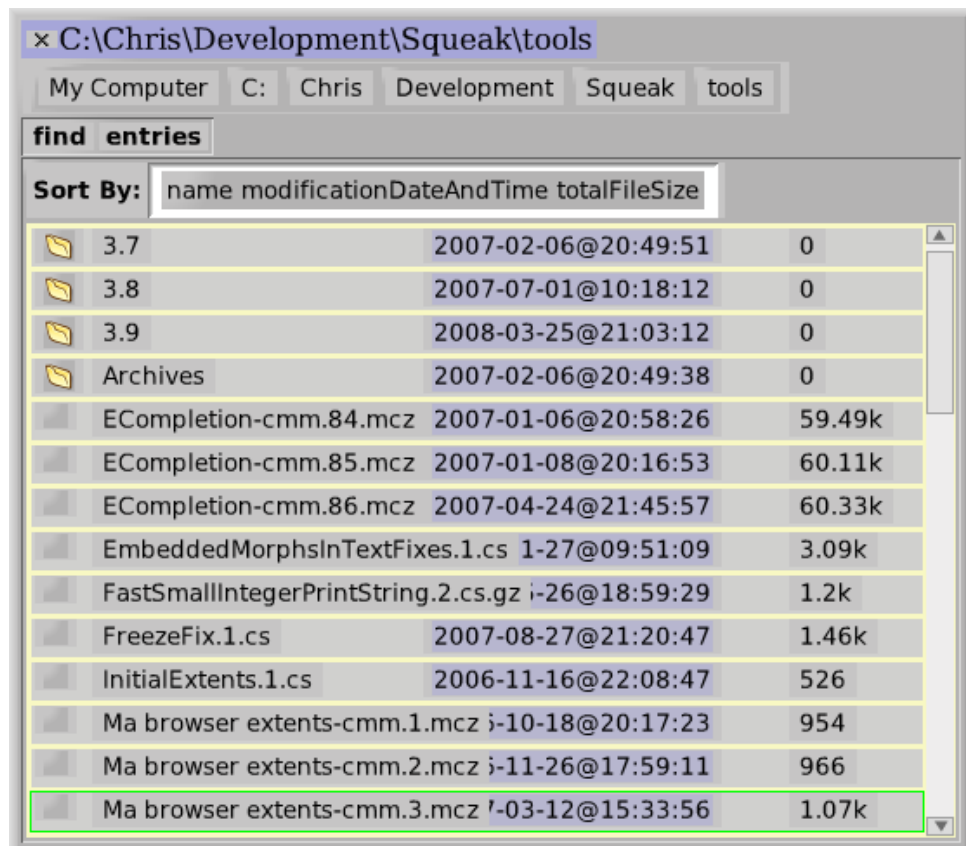
Date Picker

Dates are used often enough to offer a generic Date-picker solution. This can be used as a picker for any parameter-holder.



File Manager

Maui offers a generic file-manager.



Interesting features of the Maui FileManager:

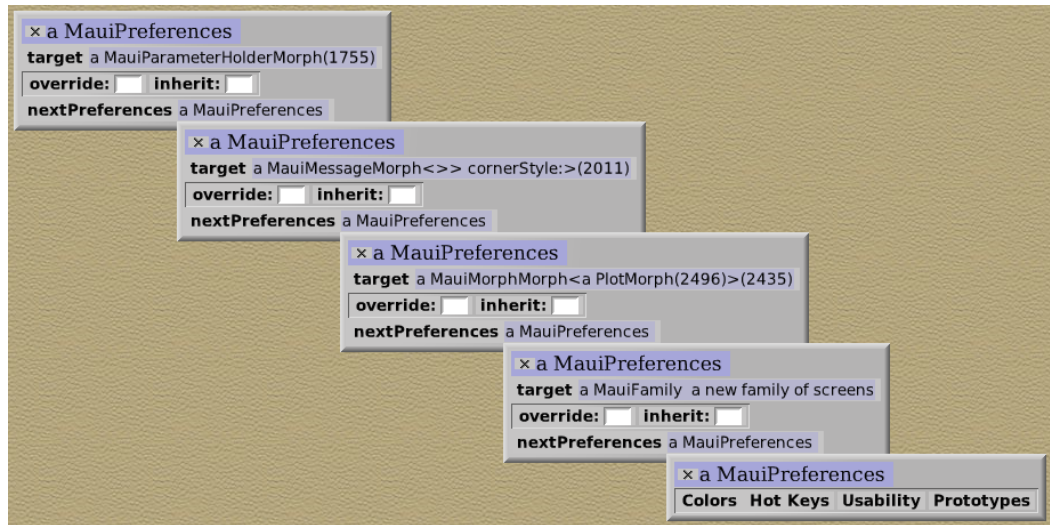
- It is a MauiDomain panel on an instance of Squeaks standard FileDirectory.
- Entries tear-off as instances of DirectoryEntry.
- DirectoryEntry's have their own custom #panel view, which renders the files initial contents.
- Files can be easily searched for by name or other attributes.
- Can show #totalSize of folders, so space-utilization allocation can be easily seen.
- All of the standard Squeak "services" are provided on the context menu.
- plus an extra service for obtaining a Maui'd FillStyle, which can then be dropped directly into MauiPreferences panel for spicing up the Maui panels themselves.

Preferences

Maui offers a broad range of user customizable preferences. There is a global MauiPreferences which can be overridden, attribute-by-attribute, by the MauiPreferences of the Family. That Preference object, in turn, is overridable by specific instances of domain objects. So, for example, a dangerous message may wish to be colored Red.

This screen shot reveals the longest-possible chain-of-responsibility of preferences. A particular parameter-holder widget defines the #target that the preferences apply to, as well as the #nextPreferences, the preference object inherited in case a particular attribute is not defined at the prior level.

The last preferences in the chain is always MauiPreferences global.



To override a particular preference, see the picker for the #override: message. To stop overriding a particular preference, use the #inherit: message.

Note, there is one MauiWorld for each Morphic World. There is one MauiFamily for each MauiWorld. By experimenting with various preferences, entire "themes" can be created at the Family / World level.

Object-Message Composition

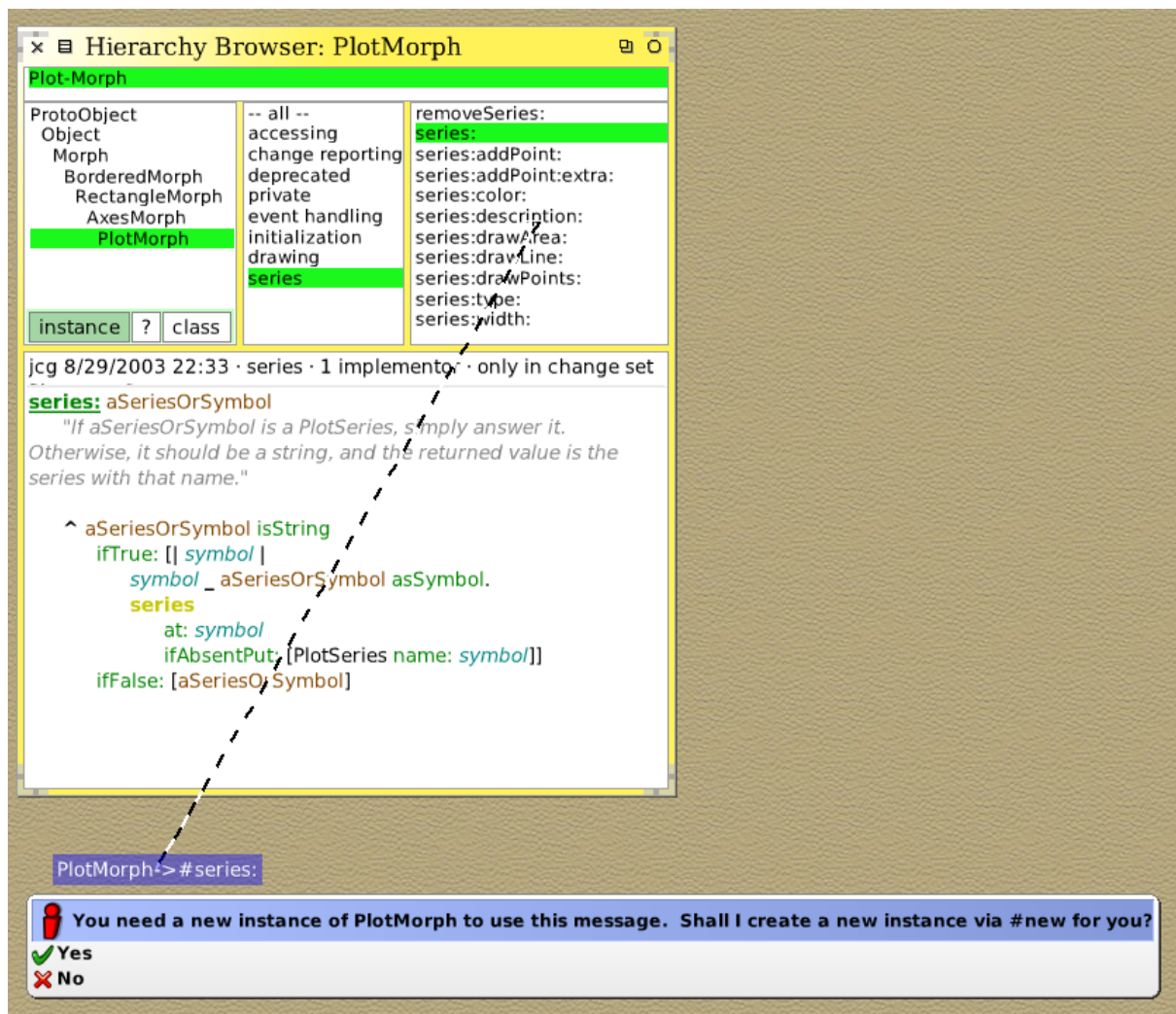
Thus far, we have discussed the least-interesting aspects of Maui; operation of its utilitarian interfaces that result from its most-interesting aspect: the quick-and-easy composition of those interfaces.

To do this, we will construct the PlotMorph view used in a prior example. If you wish to follow along as a tutorial, be sure to load the "Plot Morph" package from the SqueakMap Package Loader.

Obtaining an Object to Design

Since there is no separate "Studio", you must obtain a live object instance to design on. To do this:

- point to an existing Maui Domain object and press the lowercase 'N' command, for a new blank panel.
- send #maui to any object in a legacy Smalltalk inspector or workspace.
- drag a class or message directly out of a Smalltalk legacy browser, on to the desktop.



If you drag a Class or class-method out of the browser a Maui panel is created instantly and you will not see the above menu. However, since we are dragging an instance method, it can only be sent to an instance. Maui is prompting us if we would like to create an instance of PlotMorph via the #new message. #new is a suitable message for creating many kinds of objects, but not all. It is advisable to check the class-side API when constructing instances of unfamiliar classes.

Maui constructs a panel with that message already embedded:



About PlotMorph

A **PlotMorph** instance represents a single chart, but which may have multiple datasets plotted on it. The `#series:` method allows the creation (and subsequent access) of a **PlotSeries** instance. A **PlotSeries** represents one dataset on the chart. By hovering over the parameterholder, we see the `#series:` method takes "aSeriesOrSymbol".

If `#series:` adds a dataset, there may be a method to remove a dataset. Yes, `#removeSeries:`. Also, there must be a way to see the list of **PlotSeries** already added; yes, under the 'accessing' category there is `#series`. After dragging these messages the panel looks like this:



These messages are live, easily "invoked" by either left-clicking on the unary `#series` message or dragging or typing something into the parameter-holders for the other two.

But clicking on the `#series` message only shows, "a empty Dictionary". We want something that displays the series in a ordered list, Dictionary's are not ordered and Maui can't even display their elements without custom tooling. Only collections which can respond to `#size` and `#at:` are supported by the **MauiCollectionMorph** widget. Therefore, we need to add a method to **PlotMorph**:

```
seriesArray  
  ^ series asArray
```

Deleting an object

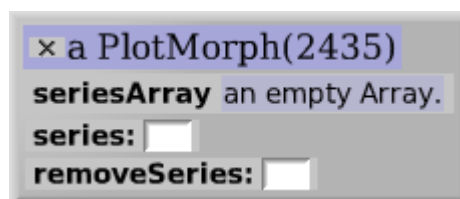
Next we want to delete the `#series` message, which answers a Dictionary, and add `#seriesArray` which answers a collection which can respond to `#size` and `#at:`. To

delete the #series message:

1. Point to it so the entire message is highlighted.
2. Press the lowercase 'X' key on the keyboard.

Pressing lowercase 'X' on any delineated object in Maui will remove it instantly with no confirmation. Warnings and confirmations are not in the nature of Maui's user-interface principle; that the user should drive the computer. If you remove the object by mistake it can be retrieved from the current MauiWorld's #clipboard.

Now the panel looks like this:

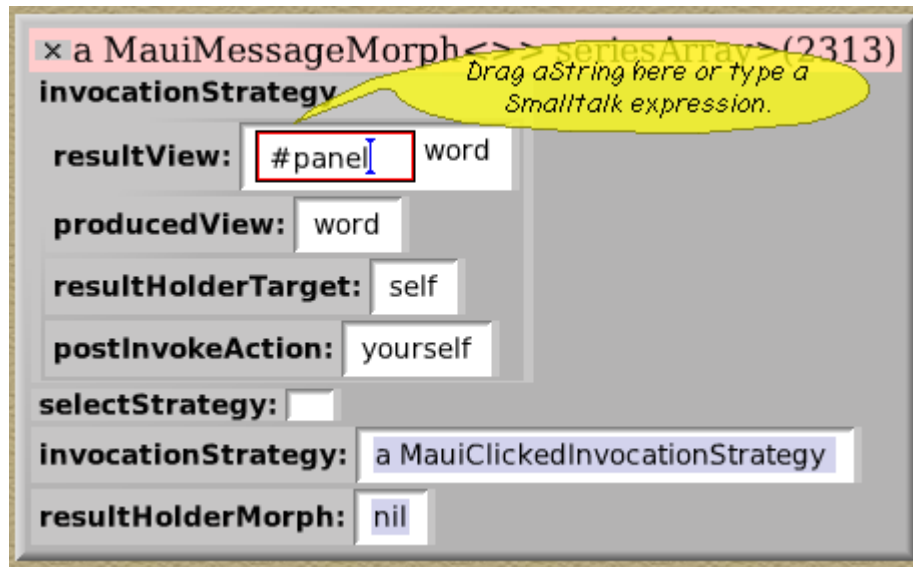


Specifying Message Settings

We need to make some improvements to that #seriesArray message. We want it to list out the series, and the simpler name, "series" is more friendly than "seriesArray". To make the improvements, open the "settings" on the message.

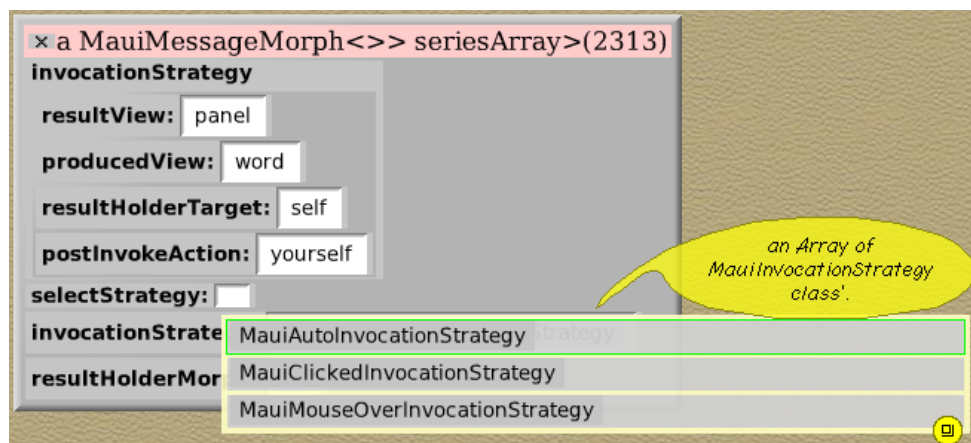
1. Point to the #seriesArray message, it is delineated.
2. Press the lowercase 'E' (for "Edit") on the keyboard.

The settings panel is displayed:



The #resultView of a MauiMessageMorph instructs Maui which view should be created to represent the **result** of invoking a message. As previously mentioned (see "Views"), there are two types of views, user-constructed, represented by Strings, and code-generated, represented by Symbols. Two code-generated views available for any object in the system are #word, the default, and #panel. We need to change the resultView: to #panel.

Next, we want the message to invoke more aggressively, so select the "Auto" invocation-strategy. Now the message will invoke automatically when the PlotMorph panel is opened rather than having to click it.



The empty Array of PlotSeries are rendered on the original panel.

Changing The Label of an object

To change the label of a the #seriesArray message (or domain morph):

1. Point at the message.
2. Press the lowercase 'L' key on the keyboard.

A dialog for adjusting the label appears. The label may be changed to something more friendly, like, "Series".

Designing the UI

From here, we could continue to add messages useful for customizing the plot. For the sake of this tutorial, let's pretend these are the requirements:

"General Options"

Ability to set the #title and background #color of the PlotMorph

"Grid Options"

Ability to set the following boolean options:

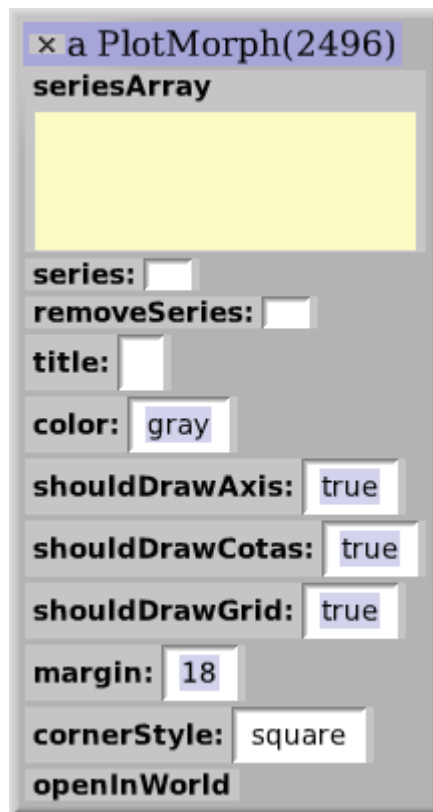
- shouldDrawAxis:
- shouldDrawCotas:
- shouldDrawGrid:

"Margin Options"

Allow specification of the margin and cornerStyle

A Quick-and-Dirty Composition

We could simple add the functionality in about 30 seconds by simply continuing to drag messages, but we might end up with something like the following:



This might be fine for some cases, but it is not very friendly because the functionality is not organized. A better way is to build panels that accomplish one task or use-case. Those finer-grained panels can then be assembled into larger super panels.

Sub-panel Composition

We will now create a set of simple panels each focused on one aspect of the PlotMorph.

Create the sub-panels

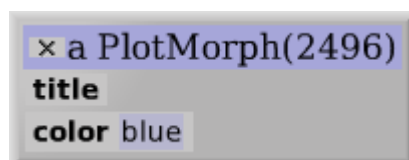
To get a new, blank view of an existing object:

1. Point at the object you wish to obtain a new view of.
2. Press the lowercase 'N' ("new blank panel") key on the keyboard.
Alternatively, the lowercase 'V' key command will produce an exact clone of that view.
3. A new panel is attached to the hand. It and the original panel are delineated because they are both just different views of the same object.
4. Press lowercase 'c' to open the class browser for this object.



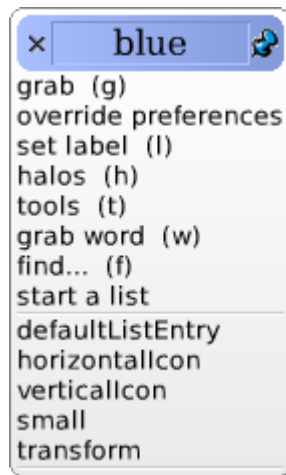
Please create the above method for this tutorial.

Set up the new sub-panel as desired by dragging the #title and #color messages. When dragging messages directly out of legacy Smalltalk browsers, it is not necessary to hold Shift like it is when dragging Maui objects around. The panel now looks like this:



Changing the resultView of a MessageMorph widget

When a message is invoked, the shape the produced result assumes, as well as where it is placed, is specified by the `#invocationStrategy` of a `MauiMessageMorph`. Let's use one of the better views available for instances of `Color`. To see the available views, simply open the context menu on the `blue` object. We see this menu:



The top portion of the menu, down to "start a list", are commands available to every `MauiMorph`. The last four items are the list of system-generated views available for instances of `Color` (the `#defaultListEntry` is a view available for all `MauiMorphs` as well). Any user-defined views of `Color` would appear below that, followed by user-defined commands for the context-menu.

Since we don't even have to do any design work to have a better compact representation of the `Color` selected for our `PlotMorph`, let's change the output of the `#color` message to `#horizontalIcon`. This is accomplished by adjusting the `#resultView` to the `#color` messages' `InvocationStrategy`.

About `InvocationStrategy`'s

To edit the `invocationStrategy` of a message:

1. Point at the message, it is highlighted.
2. Press the lowercase 'E' key (for "Edit") on the keyboard. This is the hot-key for the "settings..." option on the tools menu.

The following panel is displayed:



Look closely. On this MauiMessageMorph panel, itself, there are only four message: #invocationStrategy, selectStrategy:, #invocationStrategy:, and resultHolderMorph:.

#invocationStrategy is the getter for the MauiInvocationStrategy object of the MauiMessageMorph. See how it's output view, embedded in the #invocationStrategy message itself, has its own four messages which appears slightly indented: #resultView:, #producedView:, #resultHolderTarget:, and #postInvokeAction:, each the following function, respectively:

#resultView: is the shape the output will take. There are two types of outputs, the user-synthesized types are named by simple Strings, the computer-code synthesized types are named by selector Symbols, the default of which is usually #word.

#producedView: is the shape the "tear-off" view will take. If set to the nil, no tear-off view will be produced and, instead, the object will be picked up by the hand.

#resultHolderTarget: is a simple String which describes where the resultView is placed.

#postInvokeAction: is a selector Symbol that will be performed after the message is invoked. This was added so that when editing "lists", instances of OrderedCollection, we could send #changed to it so it would know to refresh after adding or removing elements.

selectStrategy: takes a particular Class of MauiInvocationStrategy. A new invocationStrategy instance is created. The possibilities are:

MauiAutoInvocationStrategy - Invokes automatically at opportune times, such as when the morph is initially drawn in the world, or when it changes owners, or when its underlying object receives a #changed message.

MauiClickedInvocationStrategy - Requires a left-click to invoke the message.

MauiMouseOverInvocationStrategy - Invokes once each time the hand enters the bounds of the message-morph from outside. Additionally, when the hand leaves the bounds of the result morph, it is automatically removed from the screen.

#invocationStrategy: allows a single MauiInvocationStrategy instance to be easily shared among multiple messages. Useful and powerful.

resultHolderMorph: allows a custom Maui morph to represent the the result rather than a named view. This overrides whatever is put in #resultView:.

Note, descriptions of what each message does is also presented in balloon help by hovering the hand over them.

Once the #color message's #resultView is changed to #horizontalIcon and clicking on the message once more (to re-invoke it and reproduce the result) the panel looks like this:



Making a Getter/Setter

The problem with the above is there is no way to set the title and color, which is an absolute requirement.

Many "attributes" in domain models have a "getter" and corresponding "setter". If the class has this pair of messages, Maui will make available a special menu item, "be a getter/setter". This is a shortcut for setting up the messages automatically in a very common way, appearing to handle both input and output from the same message.

To invoke this option on both the title and color messages:

1. Point at the message.
2. Select "be a getter/setter" from the tools menu (Shift + Right-click) menu.

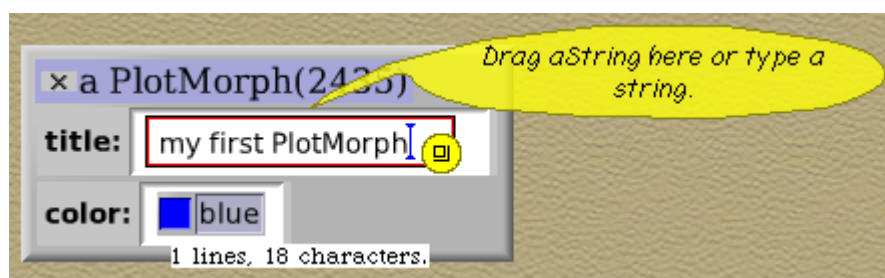
Side note: Squeak has efficient menu access, allowing items to be selected by keyboard in addition to the mouse. In this case, changing a message to a getter/setter can be accomplished this way:

1. Point at the message.

2. Press the lowercase 't' on the keyboard. The tools menu appears.
3. Press the '/' (slash) on the keyboard to filter the menu items. "Be a getter/setter" is highlighted.
4. Press Enter.

That's just four quick gestures. There tend to be quite a few getter/setters, so it can pay in the long run to commit an efficient series of gestures to muscle memory, rather than an inefficient series of gestures.

Now the attributes can be updated by supplying the parameter-holders as described in the first section.



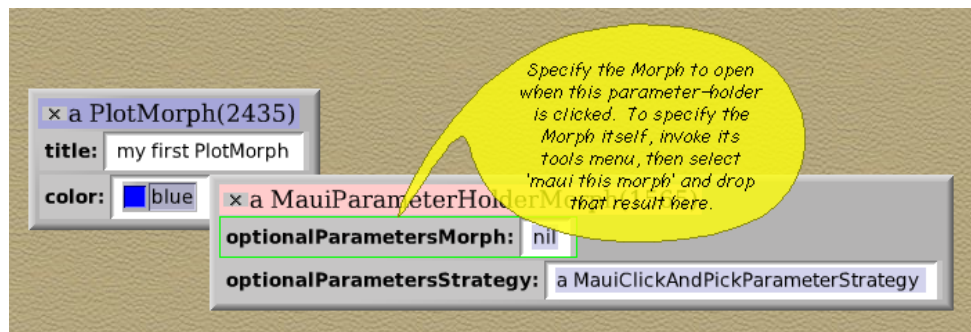
The figure shows the user updating the title (a String) without the Smalltalk single-quotes. This is because the Evaluator was changed to a MauiStringEvaluator, which was discussed in the section, "List-box pick lists".

Specifying an Object "Picker"

Although colors are available under the "Maui" tab, it may be preferable to keep them closer to the actual message needing them. To do this we specify a "picker" for the ParameterHolder widget. To do this:

1. Point at the parameter-holder for the #color: message. It is highlighted.
2. Shift + Right-click to see the design menu. Select "picker..."

The maintenance picker panel is displayed:



Maui uses "optionalParameters" terminology here. The #optionalParametersMorph: method instructions describe exactly how to set up the picker. So, let's do that:

1. Obtain a "Color picker". Maui has a special panel for the Color class which suits this purpose. Either drag one out of the "Maui" tab or evaluate the following in any text field (or workspace):

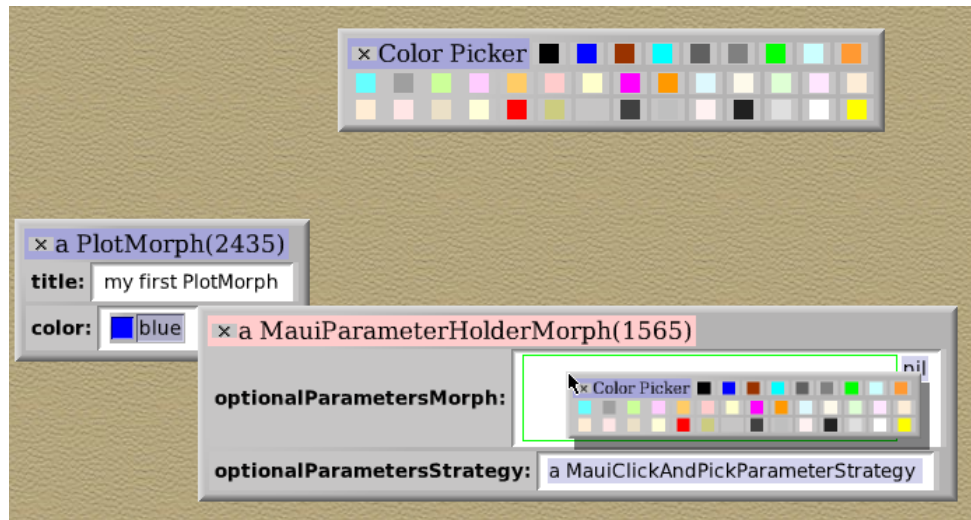
Color maui namedColors

A Color Picker is available.

1. Next, select "maui this morph" from its tools menu.



A mini "sketch" of the morph is attached to the hand. Drop it in as the argument to #optionalParametersMorph:.



Stay with the click-and-pick strategy for this picker.

What just happened is, we used a Maui panel to customize the parameter-holder of another Maui panel. We can now select colors by left-clicking directly in the parameter-holder.

The MauiBehaviorFinder

Maui integrates with the legacy Smalltalk browsers, but sometimes there are rich hierarchies of behaviors that aren't always convenient to find causing, in actual practice, panel composition to sometimes become cumbersome.

To alleviate this Maui provides its own "Behavior Finder," offering power not found in the standard Squeak system, and allow designing user-interfaces more quickly and easily. For this section, we will build the "Margin" sub-panel, which looks like this:

First, obtain a fresh blank-canvas.

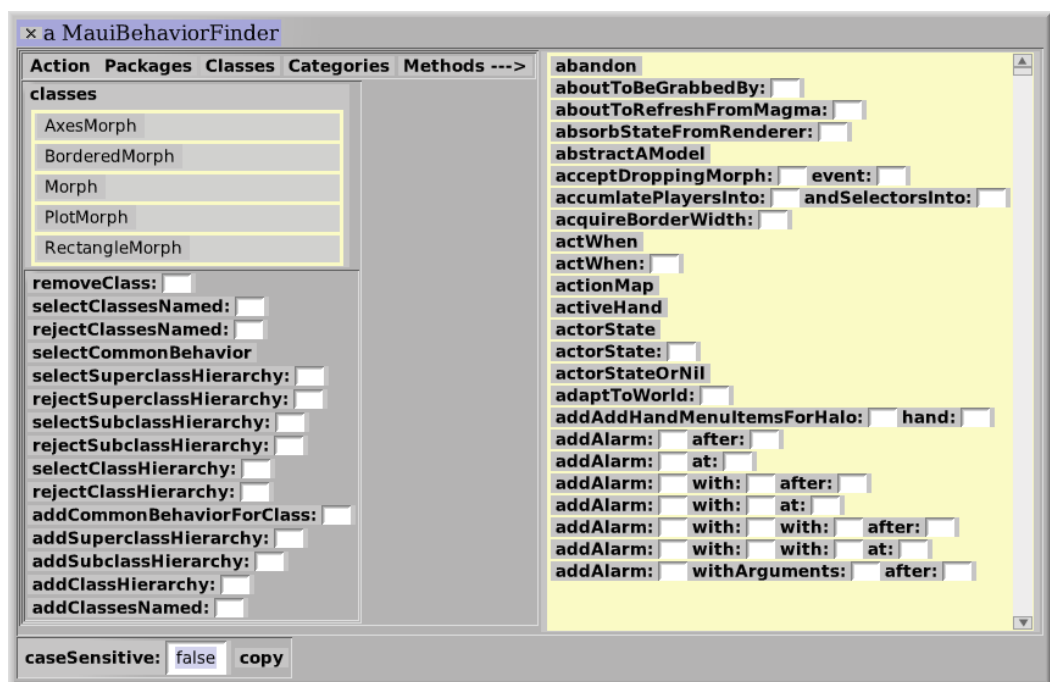
1. Point at the existing MauiMorph, it is highlighted.
2. Press the lowercase 'N' key (for new panel).

A new blank panel with for same PlotMorph domain is attached to the hand. Drop it anywhere on the desktop. The #margin: message is in AxesMorph, the #cornerStyle is all the way up in Morph. Without knowing this, designers spend quite a bit of time hunting around the browsers looking for appropriate behaviors, and Maui only allows dragging out of Class browsers, Hierarchy browsers or Package-Pane browsers, but not message-list browsers or Lexicons.

To open a MauiBehaviorFinder:

1. Point at the new panel, it is highlighted.
2. Press the lowercase 'B' command key to browse and find behavior suitable for the object.

A MauiBehaviorFinder is opened, looking something like this:



A MauiBehaviorFinder is a specialized tool for finding methods in the system. Page through the various "tabs", you will notice sets of messages that begin with "select" or "reject" and another set that begin with "add". The former remove methods from the list, the latter add. See the balloon help of each message for details. Additionally, the messages listed in the right-hand pane have the PlotMorph instance we invoked this panel for as their receiver, so messages can be "tried" right there without having to first drag them in. Very convenient!

Use the behavior finder to locate the #margin method. Now, because the message is invoked on a clicked event, we have to use another method to pick it up. How about the "clone" hot-key command:

1. Point at the #margin message.
2. Press the lowercase 'V' key on the keyboard.

A clone of the message is attached to the hand. Hold down Shift and drag it into the new panel.

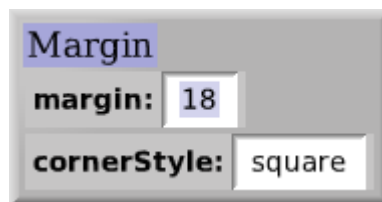
Now locate the `#cornerStyle` message and do the same. Make each one a getter/setter.

The `#cornerStyle` message takes one of two values, `#square` or `#rounded`. As an exercise, see if you can specify a "picker" for that parameter-holder that lets the user toggle between those two values. Here are the general instructions for how to do this:

1. Open the "picker..."
2. Obtain a sequenceable Collection of those two Symbols as a maui panel.
3. Select "maui this morph" on that collection panel!
4. Use that to specify the `#optionalParametersMorph`.
5. Select "a MauiCycleParametersStrategy" as the `#optionalParametersStrategy`.

Finally, change the label of the panel to "Margin".

The panel looks like this:



Completely Customized Looks

Next, we will demonstrate how any Maui panel can have a completely customized look. We will build the "Grid Options" panel with three boolean checkboxes. But instead of displaying the `#word` view of the **true** and the **false**, we will create a green checkmark and red X to represent those values.

The key concept here is that Maui Morph panels may contain not only MauiMessage morphs and other MauiDomain morphs for the same object, but also any non-Maui Morph. So, first, let's obtain the visual sketch representation; the green check and red X. Using Squeak's built-in painting tool you may produce something like this:



After selecting "Keep" we can drag it into a panel view of the **true** object. We have already covered everything needed to execute these steps, but here they are:

1. Execute "true maui" in a workspace.
2. Press the lowercase 'N' to get a new panel on it.
3. Drag the checkmark into the panel (don't forget to hold Shift).
4. Press the uppercase 'L' to remove the label bar.
5. Select "be invisible" from the "tools" menu.

This "panel" for true looks like this:

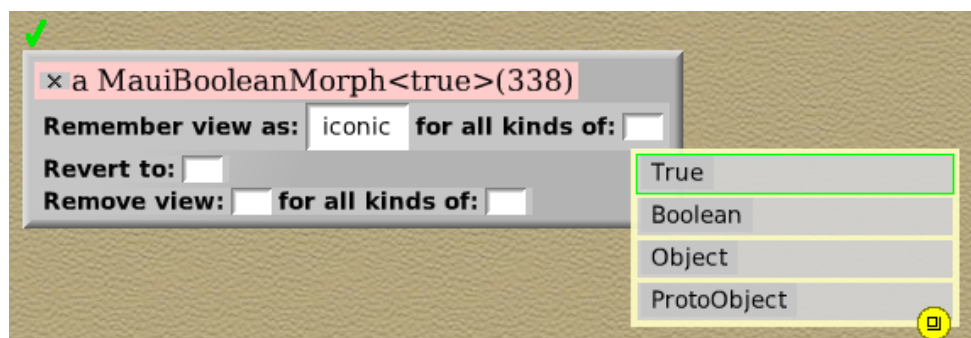


Saving Custom Panel Configurations

This was quite a few steps so let's remember this view as 'iconic'.

1. Point at the panel. Maui does not highlight in this case because basic types like true and false because it can potentially be of questionable value. Nevertheless, the object has keyboard focus.
2. Press the lowercase 'S' key on the keyboard to "Save" this panel configuration.

The "manage views..." panel is displayed.



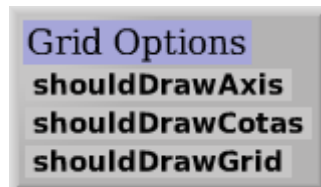
It is important to remember the view at the highest "level" possible. For this view, however, we never want anything but True instances to render as a green checkmark,

so we select True.

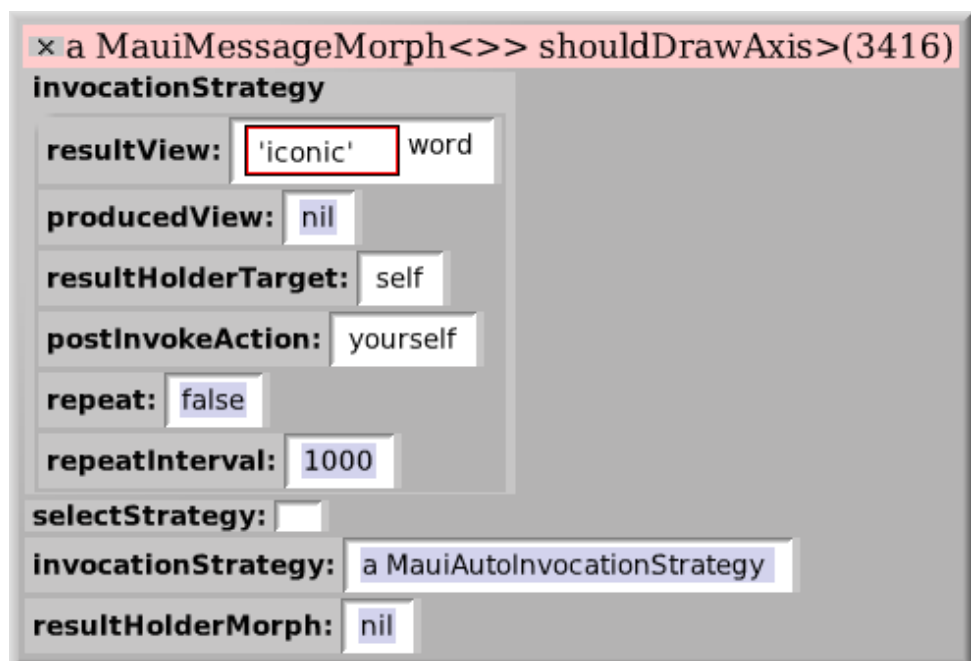
Note, now the context menu for the green checkmark includes 'iconic'.

Repeat the process for a red X, remember it also as 'iconic' at the False level.

Now we are ready to use these two new views in our "Grid Options" panel. Make a new panel, add these three messages:



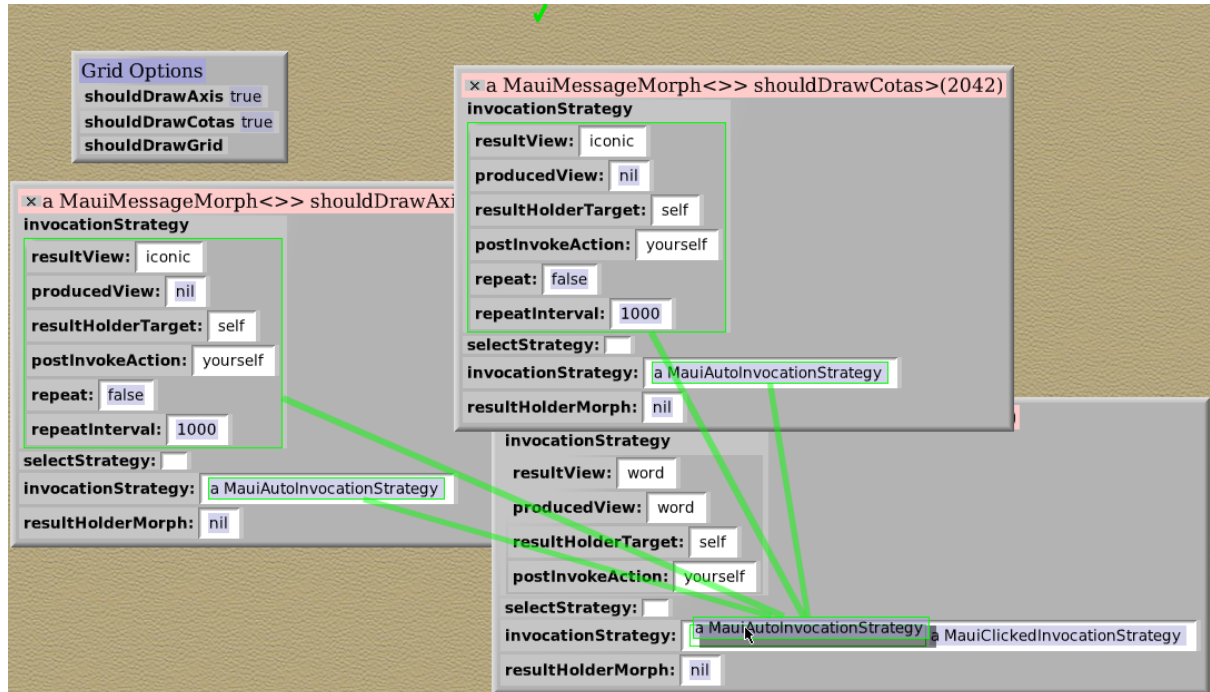
Open the InvocationStrategy for the first one, change its #resultView to 'iconic', its #producedView to nil, and use selectStrategy: to select a MauiAutoInvocationStrategy.



We need to accomplish same thing for the other two "shouldDraw..." messages, which can be done quickly by sharing the invocation strategy of the first. Specifically:

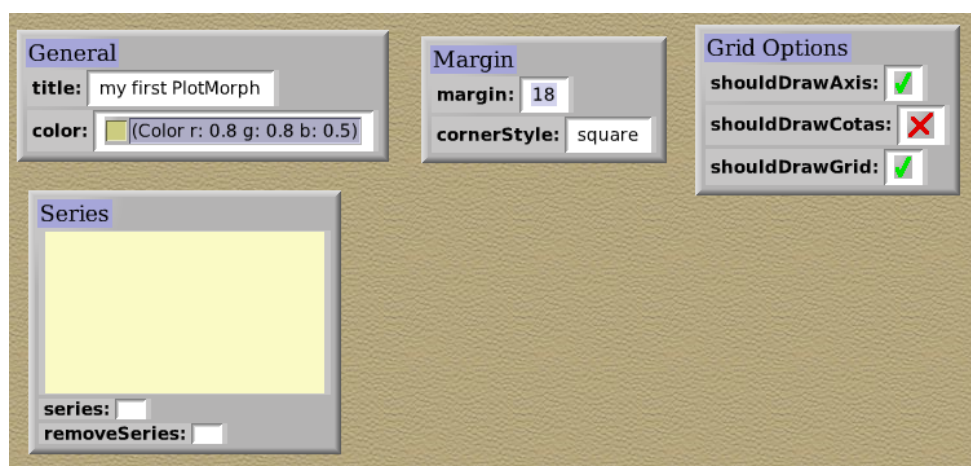
1. Point at the #shouldDrawCotas message.
2. Press the lowercase 'E' to open its settings.
3. Point at the #shouldDrawGrid message.
4. Press the lowercase 'E' to open its settings.

5. Drag the "a MauiAutoInvocationStrategy" object from the #shouldDrawAxis settings to the #invocationStrategy: setter of each of the other "shouldDraw" messages, thus:



Top off the Grid Options panel by making each message a getter/setter, making each parameter-holder behave like a checkbox, and changing its label to "Grid Options".

At this point we have four panels looking approximately like:



with various pickers or checkbox behavior for the appropriate messages.

Panels are Designed, now what?

At this point, we can do a number of different things with panels. We can:

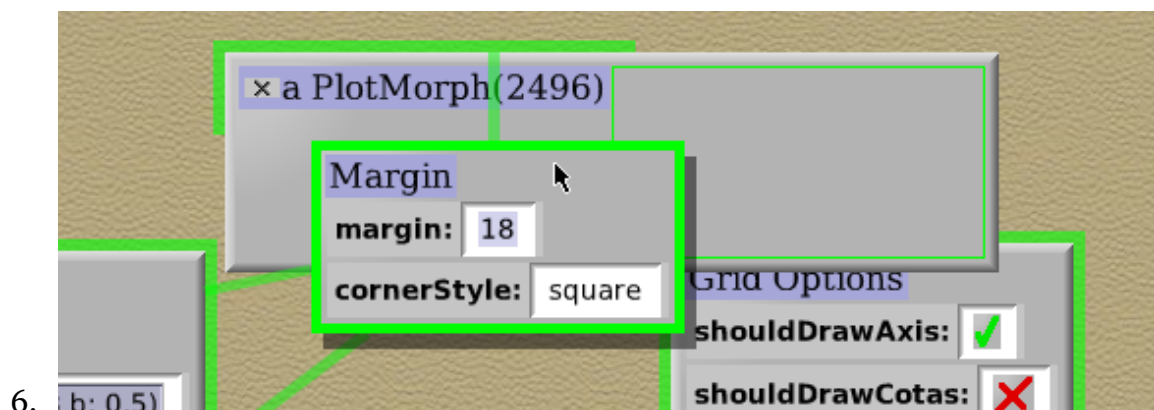
- Save each one as its own named view.
- Add each one as a "tab" of an integrated super-composition and save that view.
- Assemble them into a fixed super-composition and save that view.

Given the potential desiring high interactivity with the charts, we will choose the latter option, which trades the most screen-space for fastest and broadest scope of use. We will assemble these into a fixed super-composition.

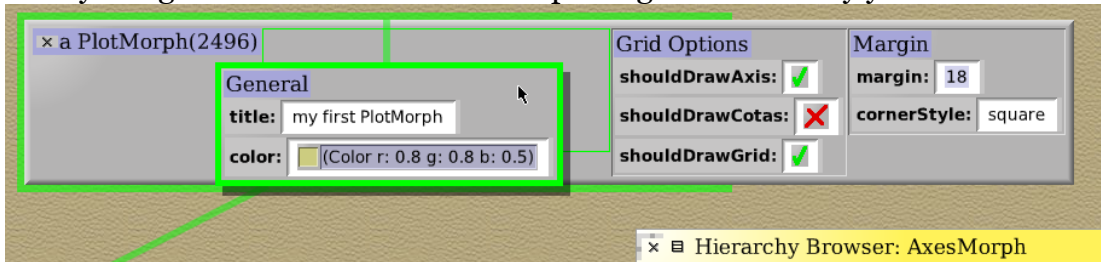
Assembling Panels

We will now assemble the panels into a single super panel. Here are the steps:

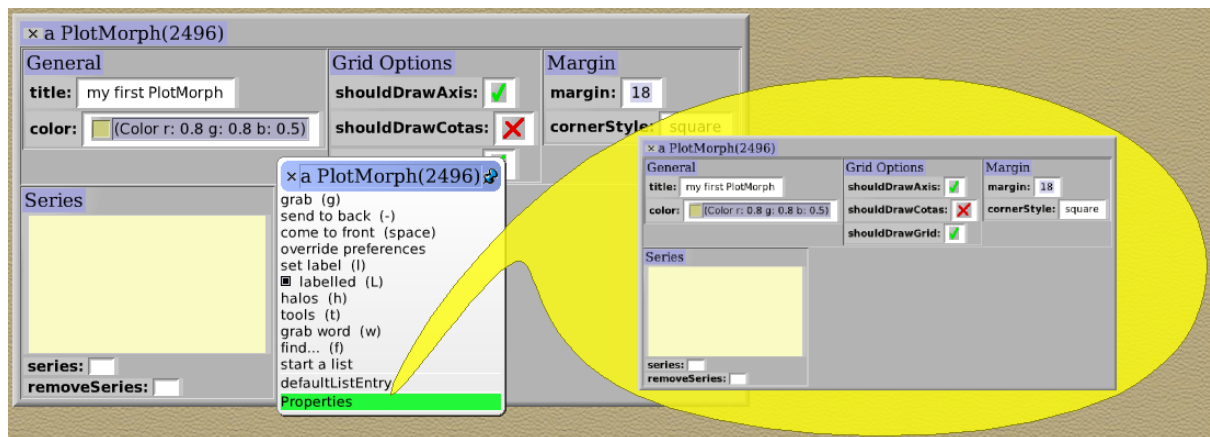
1. Obtain a new, blank panel.
2. Use the menu to change its layout to left-to-right ("tools"-->"layout"-->"arrange"-->"left-to-right").
3. Point at "Margin" panel. The entire panel, along with all the other ones, are delineated.
4. Press lowercase 'G' to grab the panel onto the hand. A left-click also accomplishes this.
5. Press and hold the Shift key as it is being dragged into the new left-to-right super panel.



7. Then the Grid Options and General panels. Sometimes panel assembly requires a little patience, deliberation and care with the mouse. It gets easier once you figure out where to make the panel grow in the way you want it.

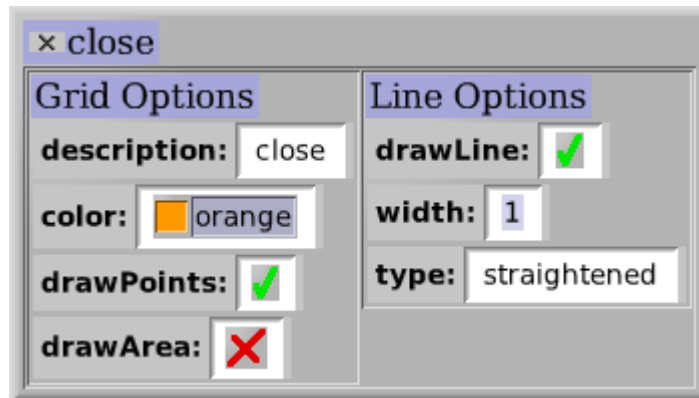


- 8.
9. The "a PlotMorph" label is not needed, remove it (point, then Shift+L). We now have the upper super-panel.
10. To maintain a reasonably squarish rectangle, we will assemble the super-panel and "Series" panel vertically into the final super-panel.
11. Obtain the final super-panel, a new blank panel (point, then press 'N' for a new view).
12. Shift+drag the "General | Grid Options | Margin" super panel and the "Series" panel into this new super panel.
13. Save the final panel as 'Properties' (point, then press 'S' to save the panel).
14. Note "Properties" appears at the bottom of the context menu for any Maui'd PlotMorph instance.



Exercise: Creating the Series Panel

As an exercise, employ the concepts used in preparing the PlotMorph to customize its PlotSeries counterpart. Designing the panel to look like this will cover most of the concepts:



To get started, you need an instance of `PlotSeries` to work on. What is the best way to get one at this point? Looking back at "Obtaining an Object to Design", the first suggestion says:

- point to an existing Maui Domain object and press the lowercase 'N' command, for a new blank panel.

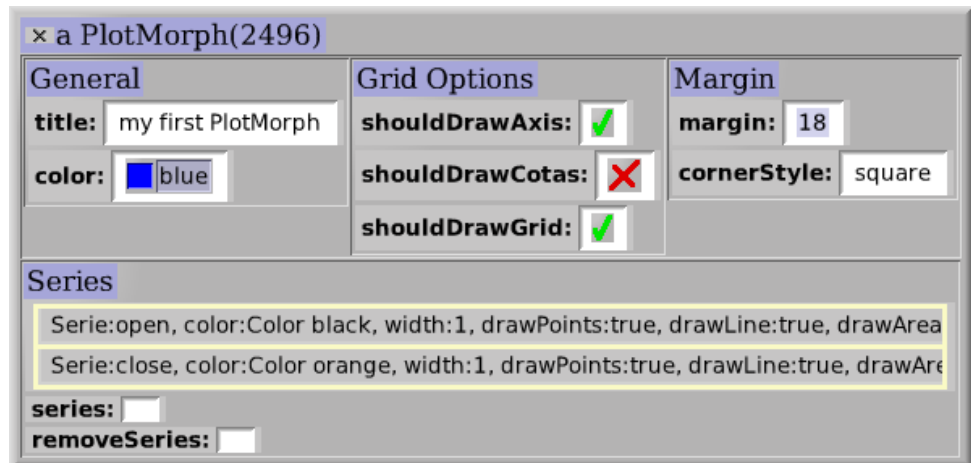
In this case, we want an instance of a `PlotSeries`, not a `PlotMorph`. To get a series, we simply invoke the `#series:` method already installed on our `PlotMorph`.

Unfortunately, `PlotMorph` does not signal `#changed` to itself, so the `#seriesArray` message must be manually refreshed (point at message, press lowercase 'R'). Then point at the `PlotSeries` in the list and press the lowercase 'N'. A new design panel.

Customizing the list-entries of a `MauiCollectionMorph`

The Default Behavior

After adding a `PlotSeries` with the `#series:` method, the entries in the list look something like this:



The entries are in a system-view called, `#defaultListEntry`. `#defaultListEntry` constructs a short-and-wide list-entry view with a proportionally-spaced layout (for columnar presentation). The default columns selected for this presentation are implemented by the message `#mauiDefaultColumns`. The default implementation of `#mauiDefaultColumns`, in Object, is simply:

```
mauiDefaultColumns
  "Answer the default selector Symbols that will make up my defaultListEntry view"
  ^ #(mauiName)
```

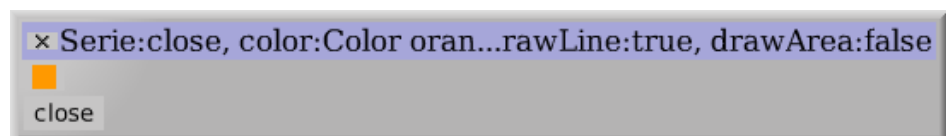
So the default-columns are just a single column containing the `#mauiName`. The default implementation of `#mauiName`, in Object simply answers its `printString`.

In this day and age, Smalltalk can do so much better than relying on `printString` to make "tables." Maui lets us do it better.

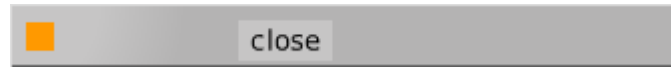
Making a Custom List-Entry

Like the getter/setter, Maui provides a helper for making views designed specifically to be entries in a list. Here is how it is used:

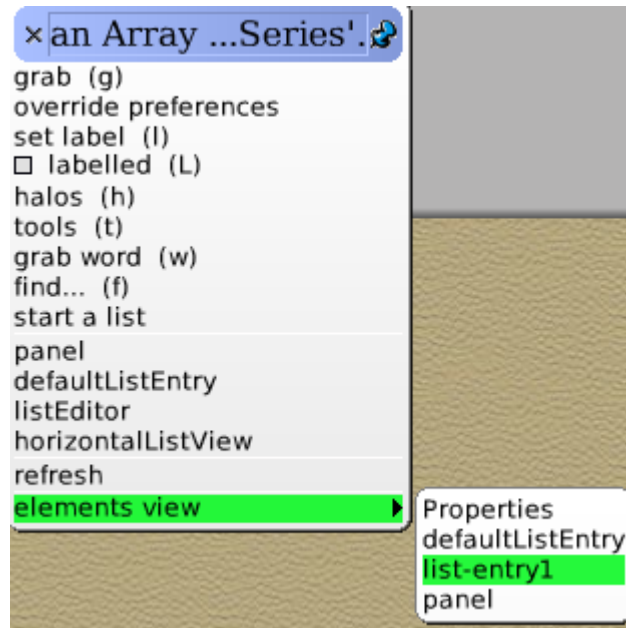
1. Obtain a new clean panel.
2. Drag in the messages you want as "columns".
3. Customize each message as desired. For example, you might want to remove their labels or change their views.



4. Select "as list-entry" from the tools menu. The view is reshaped.



5. Save the view as "list-entry1" or another name of your choice.
6. Set the "elements-view" of the **Collection** to "list-entry1" or the name you chose.



- 1.
7. Save the PlotMorph configuration once again, because we just changed its collections elements-view.

Adding Messages to the context menu

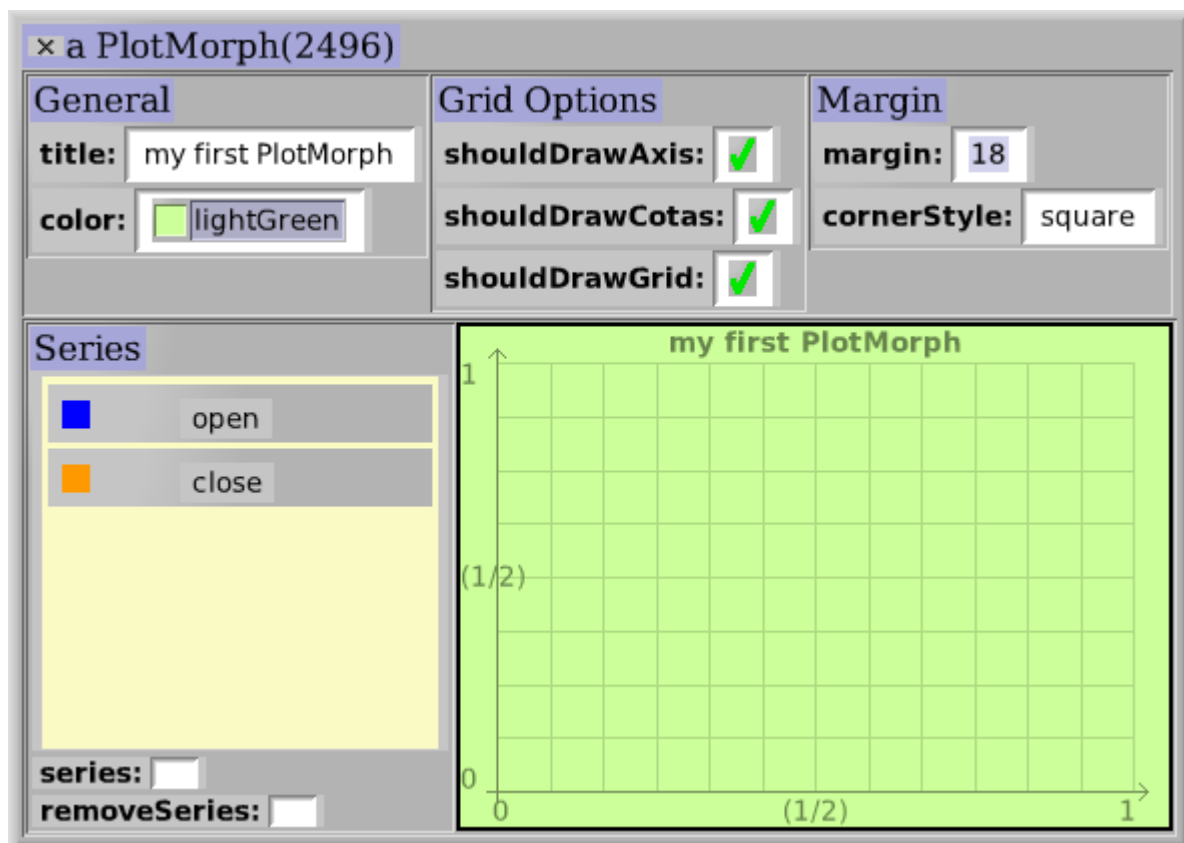
To see the graph itself, the best thing to do is to provide a #openInHand message, since that also allows it to be easily picked up if it is subsequently embedded somewhere. Dangerous or less-of-used unary messages should be placed on the context menu. That location gives them a little less accessibility and doesn't waste screen real-estate.

To add it:

- Add the #openInHand message to the panel.
- Customize the message's settings, as desired. In this case, set the resultHolderTarget: to none because we don't want the MauiMorph representation of the PlotMorph, we want the PlotMorph itself and #openInHand does that.

- Select 'add to global context-menu' or 'add to view-level context-menu' from the tools menu.
 - "add to global context-menu" will add this message to the end of the context-menu for every instance of this class of object. Other classes of objects will not have the message added.
 - "add to view-level context-menu" only adds the message to the end of the context-menu for this particular configuration of view. Don't forget to Save the view as a new version of the prototype!
- Point at the #openInHand message on the panel, press the lowercase 'X' key to delete it.
- Note the message has been added to the bottom of the standard context menu.

With this approach, the user is free to drag the PlotMorph anywhere they wish, including inside the control panel itself.



Roadmap

The biggest problem for Maui is that it is tightly-coupled to the Morphic platform of Squeak 3.9. Although powerful, there does not seem to be a lot of support from within the Squeak community for Morphic. I would like to address this in the following ways:

- Port Maui to a longer-term graphical framework. Perhaps Morphic 3.0, Tweak, or even Croquet.
- Add functionality to export the Maui prototype panels to other graphical frameworks dynamically at run-time.
- In particular, an export to a web-supported UI framework.