R3 GUI Actor overriding techniques

Author: Richard Smolak

Revision History
20-Jan-2013 ARS

Table of Contents

1. Introduction

During the R3 GUI application developement there are situations where you may look for a way how to override the default functionality of an actor in a specific style.

There are two possible techiques how to do it depending on your preferences.

2. Overriding techniques

We'll use a simple example to demonstrate both overriding techniques. Let's have the following layout:

```
view [
        field
        text-list [
                "Robert"
                "Bolek"
                "Richard"
                "Ladislav"
        ]
]
```

The code above shows a simple field and a list of names in a window. We want to use a matching filter on the names in the list every time the user types some character in the field.

2.1. Overriding actor functionality using the STYLIZE function

This techinque is useful in case you want to separate the newly added actor code from the layout dialect or if you plan to derive new styles based on the additional code.

Here is a possible solution:

```
stylize [
```

```
        filter-field: field [
                actors: [
                        on-key: [
                                ;execute the 'original' actor of FIELD style
                                do-actor/style face 'on-key arg 'field

                                ;apply the matching filter on my-list
                                val: get-face face
                                set-face/field my-list make map! either empty? val [
                                        [1 [true]]
                                ] [
                                        compose/deep [1 [find/match first value (val)]]
                                ] 'filter
                        ]
                ]
        ]
]

view [
        filter-field
        my-list: text-list [
                "Robert"
                "Bolek"
                "Richard"
                "Ladislav"
        ]
]
```

The example above works well but there is one issue. The `my-list` name is hardcoded into the definition of the new style ON-KEY actor which is not good, since it can work only in case the MY-LIST name is defined and refers to the list related to the filter field.

We should refine the example to be reusable. For that purpose we can use the `attach` dialect keyword to achieve a more general and system-friendly functionality:

```
stylize [
        filter-field: field [
                actors: [
                        on-init: [
                                ;execute the 'original' actor of FIELD style
                                do-actor/style face 'on-init arg 'field
                        ]
                        on-key: [
                                ;execute the 'original' actor of FIELD style
                                do-actor/style face 'on-key arg 'field

                                ;apply the matching filter on the attached target face(s
                                foreach target select face 'targets [
                                        if target/style = 'text-list [
                                                val: get-face face
                                                set-face/field target make map! either em
                                                        [1 [true]]
                                                ] [
                                                        compose/deep [1 [find/match firs
                                                ] 'filter
                                        ]
                                ]
                        ]
                ]
```

```
        ]
]

view [
        filter-field attach 'my-list
        my-list: text-list [
                "Robert"
                "Bolek"
                "Richard"
                "Ladislav"
        ]
]
```

The above example works with any text-list face which is attached to a filter-field. Also, the filter-field can be used as a base for new styles. Notice the code used in `stylize` is also longer and more complex, because in fact we derived a new child style from the FIELD parent style.

2.2. Inline overriding using the layout dialect

Sometimes you don’t want to create a new derived style. This can be handy in the following conditions:

- you intend to use the changed behaviour just once, and
- don’t want to use the changed actor behaviour as a base for new styles

Here is a possible solution:

```
match-filter: func [
        list [object!]
        value [string!]
] [
        ;apply the matching filter on my-list
        set-face/field list make map! either empty? value [
                [1 [true]]
        ] [
                compose/deep [1 [find/match first value (value)]]
        ] 'filter
]

view [
        field on-key [
                ;execute the 'original' actor of the field
                do-actor/style face 'on-key arg 'field
                ;call the filtering function
                match-filter my-list get-face face
        ]
        my-list: text-list [
                "Robert"
                "Bolek"
                "Richard"
                "Ladislav"
        ]
]
```

In the above code the ON-KEY word in the layout dialect means you want to override the on-key actor of the FIELD style, i.e., instead of the original actor, the given code will be evaluated.

You can override any other existing actor using the same approach. The actor name is then followed by a block which is used instead of the original ON-KEY actor block defined in the FIELD style.

In this case we don’t mind whether the inline block contains any application specific code.

Same as with the original style actor, you are free to put any code into the inline actor block.

For example, you can call the original actor before or after your application-dependent code. Or even don’t call the original style actor at all or chain multiple actor calls and so on. It all depends on your needs.