MATCH SETTINGS HIERARCHY

In a simple case of image matching you can only define the path to the needle and it will be performed for you. Example code for matching an image */path/to/image_folder/existing-image* is:

```
guibender.add_image_path('/path/to/image_folder')
guibender.find('existing-image')
```

However, one main disadvantage of this is that this matching requires high default similarity and can be really fragile even if the slightest change in the GUI occurs. You can of course use the calibrator to find the match settings for highest possible similarity in the new case but much better would be if you understand a little bit more about how the match settings work and how modifying them a tiny bit can help you achieve large flexibility of the virtuser scriptlets and robustness to insignificant changes in the GUI. So let's get started.

Next to any image file *existing-image.format* you can have a file to contain its match settings with the name *existing-image.match*. If this file exists the match settings from it are automatically extracted and used during the matching process of the respective image. Of course there might be some cases where you need more specific match settings so the match settings can always be processed further in the virtuser scriptlets you develop. Overall, this file ensures that you have reliable long term settings for matching the specific image. The file however is specific for just one image. For all files without such specific match settings, the defaults from the *Settings* object are used. So all in all the hierarchy of match settings scopes and overriding is the following:

```
Settings() class - default settings for all images every time
  -> "image.match" file - default settings for a specific image every time
    -> image.match_settings attribute - settings for a specific image object
(loaded image)
      -> region.imagefinder.eq attribute - settings for a specific region
```

This allows for a kind of inheritance of the match settings from the most general to the most specific match cases. It is really necessary since every time you match a needle image, you do this with respect to different haystack image where there might be different "distractions" relative to the actual match.

So to provide more details about the way to create more durable match settings or temporary such we need to know the relevant API of the classes *Settings*, *Image*, *Region*, and *CVEqualizer*. Every time you perform matching you have to create Image object from the image name (provided you have added the path to the image directory containing it) and use that object as a needle image during matching. Of course you can use that *Image* object multiple times in which case its last and most current matching settings will be reused. If no match file was provided the match settings will be the ones used for any matching and extracted from the *Settings* class. Let's say that you need better match settings for that specific image. For example if you want to change the similarity required during each matching of this loaded image you should do the following:

```
new_similarity = 0.5
image = Image('existing-image')
```

```
image.use_own_settings = True
image.match_settings.p["find"]["similarity"].value = new_similarity
# alternatively you can use a simplified API but only for a few basic parameters
# image.use_own_settings = True
# new_image = image.with_similarity(new_similarity)
```

We will explain more about the structure of the match settings a bit later, this is just an example in order to see how to use the *match_settings* attribute of the loaded image object. The new match settings will be used in every matching where this *Image* object is provided. You can be even more specific and change the match settings for a single matching of the *Image* object (using *Region* as a more ephemeral object than *Image*) by doing:

```
new_similarity = 0.5
region = Region()
region.imagefinder.eq.p["find"]["similarity"].value = new_similarity
match = region.find('existing-image')
```

Notice however that in the long term this becomes cumbersome since you would have to set change the match settings each time you load the image or define a region. The way to solve this is to give more generality and durability to the performed changes by saving them as a match file. To do so use this:

```
new_similarity = 0.5
image = Image('existing-image')
image.match_settings.p["find"]["similarity"].value = new_similarity
image.use_own_settings = True
returned_image = image.save('/path/to/image_folder/existing-image.format')
```

This will additionally save a match file with the match settings of the image which will then be detected every next time you load the image:

```
image = Image('existing-image')
print image.use_own_settings
# True
print image.match_settings.p["find"]["similarity"].value
# 0.5
```

Are we cool? All you need now is to know a bit more about the structure of the match settings and you are good to go.

MATCH SETTINGS STRUCTURE

The match settings are simply a *CVEqualizer* object. Knowing that, we can play with it a lot and go beyond the hierarchy explained above by using custom match settings for multiple images and more. To create a few *Image* objects with the same match settings you can do:

```
new_similarity = 0.5
```

```
match_settings = CVEqualizer()
match_settings.p["find"]["similarity"].value = new_similarity
image = Image('existing-image', match_settings=match_settings)
image2 = Image('existing-image2', match_settings=match_settings)
```

The general scientific field of computer vision contains many different algorithms with algorithm specific parameters that can be used to perform image matching. Therefore all its complexity is captured and contained in this *CVEqualizer* object. The used algorithms are called backends and have five main categories:

- ➔ **find** - the most general backend category defining which of the other categories will be used
  1. **template** - use template matching and more specifically the backend defined in *tmatch*
  2. **feature** - use feature matching and more specifically the combination of backend algorithms defined in *fdetect*, *fextract*, and *fmatch*
  3. **hybrid** - use a mixture of both that we developed to all their advantages
- ➔ **tmatch** - template matchers
- ➔ **fdetect** - feature detectors
- ➔ **fextract** - feature extractors
- ➔ **fmatch** - feature matchers

To define a set of backend algorithms that you would like to use in matching settings do:

```
match_settings = CVEqualizer()
match_settings.set_backend("find", "feature")
```

OR

```
match_settings = CVEqualizer()
match_settings.configure_backend(find_image = "feature", template_match = None,
                                 feature_detect = None, feature_extract = None,
                                 feature_match = None)
```

OR

```
region = Region()
region.configure_find(find_image = "feature")
match = region.find('existing-image')
```

Notice that if you replace a backend algorithm all relevant parameters will be also replaced and if you set the same backend algorithm again all relevant parameters will be set to the default. All parameters relevant to some backend are stored in the same category and can be accessed in a similar way from the parameters attribute of the *CVEqualizer* object. All general parameters are stored in the most general backend directory *find*. Thus, to access the similarity we used

```
print match_settings.p["find"]["similarity"].value
```

because it is a general parameter. To access the *nFeatures* parameter of the ORB feature detector backend first set it as backend algorithm for feature detection and then it should be available:

```
match_settings = CVEqualizer()
match_settings.configure_backend(find_image = "feature",
                                 feature_detect = "ORB")
print match_settings.p["fdetect"]["nFeatures"].value
```

Lastly, each parameter is a bit more complex than just a value. The relevant attributes of a parameter are visible in any match file (which of course is completely readable config file):

```
nFeatures = <value='500' min='None' max='None' delta='1' tolerance='0.9'
fixed='True'>
```

From *min* and *max* we construct *range* tuple for the parameter object which looks like

```
match.settings.p["fdetect"]["nFeatures"].range = (min, max)
```

If you don't care about calibration you don't need to know anything about the other parameters. The *delta*, *tolerance*, and *fixed* attributes are used by the calibrator where fixed would make the calibrator skip calibrating the parameter. The other two would define how much guessing will be used and when will the parameter have an acceptable value for the calibrator.

The complexity behind the match settings is a trade-off for the availability of all computer vision and more specifically OpenCV methods and the power it could give you when performing image matching. You need to read more about the specific computer vision algorithm to know more about the specific backend parameters - this tutorial is just to show you how to modify them and use the best methods you have found for one or all of your images.