HOWTO - Developing new components

Developing new components to the pipeline is not a difficult task, but a protocol has to be followed. First, all parameters for executing the component should be placed in a configuration file, no command line parameters are allowed, and the component should follow a specific calling protocol. Second, all sequence input, output and manipulation should be performed using the interface of the `SequenceObject` Perl modules. This document provides some documentation on how to write new EGene components. If you are interested in developing new components, please contact the authors.

Calling protocol

All components are called using the command line below:

```
<component_name> --conf=<configuration file name>
```

The component should NOT put any data in the standard output, as STDOUT is used for communication between components. You are free to use the STDERR.

The configuration file lines should have the following format:

```
<parameter_name> = <parameter_value>
```

To make things easier for the end user, you should let the order of the parameters be free.

To facilitate the creation of Perl components following the configuration file and calling protocol conventions, we provide a small Perl script, `generate_perl_header.pl`. This script reads a specification that should contain all the parameters of configuration files of the target component, and produces a "skeleton" Perl program that works in the required manner (including the command line format and order-free parameter reading). A sample input for this component would be:

```
==file_name
==path
user_id=
protocol="html"
```

Each line of the specification above describes an entry in the configuration files of the component. Mandatory arguments should be preceded by a double equal sign ("=="), optional arguments should be followed by one equal sign ("="), and, eventually, a default value. The skeleton generated for the input above will expect configuration files with two mandatory arguments (`file_name` and `path`), and two optional arguments (`user_id` and `protocol`). The `protocol` argument, if not specified, is always "`html`". No typing is enforced on the parameters.

The code specified in the skeleton program opens the configuration file, checks if the mandatory arguments are present (aborting otherwise), and fills local variables with the values specified in the configuration files. The local variables have the same name as the corresponding configuration arguments (e.g. `$protocol`, `$file_name`, `$path`,`$user_id`).

The auxiliary program, called `generate_perl_header.pl`, reads input from STDIN and outputs the results into STDOUT. It can be downloaded from EGene's web site (http://www.lbm.fmvz.usp.br/egene). We list below the result of running `generate_perl_header.pl` to the input above.

```
#-------------------------------------------------------------
#!/usr/bin/perl
use strict;
use Getopt::Long;
use EGene::SequenceObject;
#mandatory fields
my $file_name;
my $path;
#optional arguments and configuration defaults
my $user_id;
my $protocol="html";
#local variables
my @arguments;
my %config;
my $configFile;
my $missingArgument=0;
my $conf;
#read configuration file
GetOptions ("conf=s" => \$conf);
open(CONFIG, "< $conf") or die("missing configuration file
$configFile\n");
my $configLine;
while ($configLine = <CONFIG>)
  $configLine =~ s/^\s+//;
  $configLine =~ s/\s+\Z//;
  $configLine =~ s/\s+\=/\=/;
  $configLine =~ s/\=\s+/\=/;
  if ($configLine =~ /^\#/ || !($configLine =~ /(.)\=(.)/))
    next;
  }
  chomp $configLine;
  @arguments = split("=", $configLine);
  $config$arguments[0]}=$arguments[1];
}

close(CONFIG);
#now check if any mandatory argument is missing
    if (!exists($config"file_name"}))
        $missingArgument = 1;
        print "Missing mandatory configuration
argument:file_name\n";
    }
    else $file_name = $config"file_name"}};
    if (!exists($config"path"}))
        $missingArgument = 1;
        print "Missing mandatory configuration
argument:path\n";
    }
    else $path = $config"path"}};
```

```
 if ($missingArgument)
     die "\n\nCannot run program, mandatory configuration
argument missing (see above)\
 "
 }
 #set optional arguments that were declared in configuration
file
     if (defined($config"user_id"}))
        $user_id = $config"user_id"};
     }
     if (defined($config"protocol"}))
        $protocol = $config"protocol"};
     }
 #-------------------------------------------------------------
```

Internal structure of components

To ensure the component works in a pipeline manner, abstracting, input and output format, the following loop skeleton should be used:

```
my $sequence_object = new SequenceObject;
while ($sequence_object->read())
...process data...
...update sequence data...
$sequence_object->print();
}
...clean up temporary files and directories...
```

The script can freely manipulate data internally, however, to record results that should be stored in the sequence representation the interface specified in `SequenceObject.pm` must be used. Sequences are stored with a record of all operations performed on them. The interface allows the components also to record all programs that were applied to each sequence. These runs are recorded as logs. Logs and operations can be used later to build reports.

`SequenceObject` interface

`SequenceObject` plays an essential role in the portability of representations. You should not modify it on your own at the risk of incompatibility with future versions of EGene. If you need to enrich the interface, you should first contact the authors.

NOTE: the interface soon will be greatly expanded to include annotation information

Currently the interface includes the following functions:

`initialize_from_xml` – receives an XML string and initializes the object.

`initialize_from_phd` – receives a PHD representation in a string and initializes the object.

`xml_string` – returns a string with the XML representation of the object.

`phd_string` – returns a string with the PHD representation of the string.

masked_sequence – returns a string with all the bases of the sequence, where masked bases are substituted for "X".

current_sequence – returns a string with the bases of the sequence after trimming.

sequence – returns a string with all the original bases of the sequence.

current_qual_vector – returns a vector with the quality values of the current sequence.

qual_vector - returns a vector with **all** quality values of the original sequence.

location_vector - returns all the location values that were present in the PHD original file, if it is the case.

file_name_header – returns a string identifying the sequence that can be used as a file name (no spaces).

print – sends the sequence data to the output.

read – reads sequence data from the pipeline.

sequence_name – returns a string with the name of the sequence.

log - creates a log entry in the sequence. This function is used by components to record data about the processing, as parameters used and date. This function receives the log string and a log key and returns a log number. Log numbers should be used when calling functions that alter the sequence contents, so `SequenceObject` can keep track of what changes were made by which programs.

trim_left – trim a specified number of bases from the 5' end of the sequence. A log number can also be provided.

trim_right – same for the 3' end.

mask_current – masks some region of *current sequence*, a log number can also be provided.

mask – same as above, but position is in original sequence.

invalidate – marks sequence as invalid, a log should be provided.

get_entries_for_masking – returns an array with all logs associated with masking operations.

get_entries_for_trimming – same for trimming operations.

get_entries_for_filtering – same for invalidation operations.

get_entries_for_key – returns all logs associated with a specific key.

get_program - returns a string with the parameters used for running a program associated with a specific log number. It is used in report components, in particular in the complete graphical report, to retrieve information about masking and invalidation.

fasta_header – returns a string that can be used as a FASTA header.

is_valid - checks if a sequence is valid.

bases_trimmed_right – returns the number of bases trimmed from the 3' end.

bases_trimmed_left - same for 5' end.