Dev manual for Toggle

Version 2.0

Authoring by Toggle dev Team

April 19, 2017

Contents

Chapter 1

Introduction

1.1 To whom this manual is addressed

The current manual is addressed to new TOGGLE developers, *i.e.* people wanting to develop new tools in the TOGGLE framework. If you just want to use already existing TOGGLE bricks, you do not need to read it, you can go directly to the user manual on the github of the project.

1.2 General things about the TOGGLE github

Developers are required to work on the TOGGLE-dev github, accessible at `https://github.com/SouthGreenPlatform/TOGGLE-DEV`.

1.2.1 Preparing your working environment

You first have to clone the TOGGLE-DEV current version using the following command

```
#Cloning
git clone https://github.com/SouthGreenPlatform/TOGGLE-DEV /path/for/cloning

#Moving to the cloned folder
cd /path/for/cloning
```

Then, create your own development branch using the following commands

```
#Create a branch
git branch branchName

#Switch to this branch
git checkout branchName

#Make a change then perform the fist commit
git commit -m "My comment" changedFile
```

```
#Push this local branch to GitHub
git push https://github.com/SouthGreenPlatform/TOGGLE-DEV.git branchName
```

This will prevent any regression in the current version and thus allow a reliable development.

Integration of new branchs will be perfomed by power users under request on the github. The integration depends on the correct application of the following recommandations, especially tests.

1.3 General things about the conventions and nomenclatures in TOGGLE

In TOGGLE, the nomenclature is quite the same for all filenames, variables, modules or functions. The way we will name a variable representing the output BAM file e.g. is **bamOutput**, thus all in lowercases, upper case being used to separate words. A multiple words function such as the picard-tools CreateSequenceDictionary one will thus be **picardToolsCreateSequenceDictionary**.

Chapter 2

Creating a new module

A module is a set of functions related to each others, either because they came from the same software suite (`gatk.pm`,`bwa.pm`,...), or that they impact the same types of files (`fastqUtils.pm`).

2.1 Names

The name of the *Perl* module must be explicite. Do not use weird names such as "myTestModule.pm" to publish on the GitHub. Generally the name is related to the function target (software or format).

2.2 Requirements and Declaration

All modules created for TOGGLE must be structuredas follows, with the same preamble:

```
1  package myName;
2
3  #
       ##############################################################################
4  #
5  # Copyright 2014-2015 IRD-CIRAD-INRA-ADNid
6  #
7  # This program is free software; you can redistribute it and/
       or modify
8  # it under the terms of the GNU General Public License as
       published by
9  # the Free Software Foundation; either version 3 of the
       License, or
10 # (at your option) any later version.
11 #
12 # This program is distributed in the hope that it will be
       useful,
13 # but WITHOUT ANY WARRANTY; without even the implied warranty
       of
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
15 # GNU General Public License for more details.
16 #
17 # You should have received a copy of the GNU General Public
     License
18 # along with this program; if not, see <http://www.gnu.org/
     licenses/> or
19 # write to the Free Software Foundation, Inc.,
20 # 51 Franklin Street, Fifth Floor, Boston,
21 # MA 02110-1301, USA.
22 #
23 # You should have received a copy of the CeCILL-C license with
      this program.
24 #If not see <http://www.cecill.info/licences/Licence_CeCILL-
     C_V1-en.txt>
25 #
26 # Intellectual property belongs to IRD, CIRAD and South Green
     developpement plateform for all versions also for ADNid
     for v2 and v3 and INRA for v3
27 # Version 1 written by Cecile Monat, Ayite Kougbeadjo,
     Christine Tranchant, Cedric Farcy, Mawusse Agbessi,
     Maryline Summo, and Francois Sabot
28 # Version 2 written by Cecile Monat, Christine Tranchant,
     Cedric Farcy, Enrique Ortega-Abboud, Julie Orjuela-Bouniol
     , Sebastien Ravel, Souhila Amanzougarene, and Francois
     Sabot
29 # Version 3 written by Cecile Monat, Christine Tranchant,
     Laura Helou, Abdoulaye Diallo, Julie Orjuela-Bouniol,
     Sebastien Ravel, Gautier Sarah, and Francois Sabot
30 #
31 #
     ##############################################################################
32
33 use strict;
34 use warnings;
35 use localConfig;
36 use toolbox;
37
38  sub foo{}
39
40  sub bar{}
41
42  1;
43
```

The licence must be conserved as given, except for an addition of the current developer name and institute.

The `use` lines are also mandatory to have access to the `toolbox` function (`run`,...), as described latter, as well as to the softwares location (`localConfig.pm` module).

Chapter 3

Creating a new function

Here is an example of a currently developed function

```
1
2  ##SAMTOOLS SORT
3  #Sort alignments by leftmost coordinates.
4  sub samToolsSort{
5       my($bamFileIn,$optionsHachees)=@_;
6       if (toolbox::sizeFile($bamFileIn)==1){ ##Check if entry
       file exist and is not empty
7
8             #Check if the format is correct
9             if (toolbox::checkSamOrBamFormat($bamFileIn)==0) {#
       The file is not a BAM/SAM file
10                  toolbox::exportLog("ERROR: samTools::
       samToolsSort : The file $bamFileIn is not a SAM/BAM file\n
       ",0);
11                  return 0;
12            }
13
14            my $bamPath=toolbox::extractPath($bamFileIn);
15            my $bamFileName=toolbox::extractName($bamFileIn).".
       SAMTOOLSSORT";
16            my $bamFileOut = $bamPath.$bamFileName;
17            my $options="";
18            if ($optionsHachees) {
19                  $options=toolbox::extractOptions(
       $optionsHachees);
20            }
21            my $command=$samtools." sort ".$options." ".
       $bamFileIn." ".$bamFileOut;
22            #Execute command
23            if(toolbox::run($command)==1){
24                  toolbox::exportLog("INFOS: samTools::
       samToolsSort : Correctly done\n",1);
25                  return 1;#Command Ok
26            }else{
27                        toolbox::exportLog("ERROR: samTools::
       samToolsSort : Uncorrectly done\n",0);
```

```
28                      return 0;#Command not Ok
29                  }
30      }else{
31          toolbox::exportLog("ERROR: samTools::samToolsSort :
    The file $bamFileIn is uncorrect\n",0);
32          return 0;#File not Ok
33      }
34  }
```

As you can see, the code is quite well structured and commented for a *Perl* code...

Moreover, *ALL THE FUNCTIONS MUST UNITARY*, i.e. the shortest possible.

All system calls will be performed throught the `toolbox::run` function

3.1 Nomenclature, Indentation and commentaries

As explained earlier, the names of variables and functions must be **functionName**.

Indentation is mandatory, as well as commentaries.

3.2 Basic structure of the function

A function will be designed as follow:

1. Picking up input data, output data (if any) and options
2. Verifying the input format, if any
3. Creating the output file name if not supplied already
4. Picking up the options in a text format (using `toolbox::extractOptions` function
5. Creating the command line
6. Sending command line to `toolbox::run` using a *if*
7. Sending log to `toolbox::exportLog` function

3.3 The `toolbox::exportLog` and `toolbox::run` functions

3.3.1 `toolbox::exportLog`

`toolbox::exportLog` is a intreasic feature in TOGGLE that will fill the various logs all along the pipeline running.

In a basic way, you can send any message to the current logs. INFOS and WARNING messages will not kill the current process, while ERRO will.

To construct a message, please follow the current nomenclature

INFOS : toolbox::exportLog("INFOS: myModule::myFunction : Coffee is ready",1);

WARNINGS : toolbox::exportLog("WARNING: myModule::myFunction : Coffee is not ready yet",2);

ERRORS : toolbox::exportLog("ERROR: myModule::myFunction : No coffee left!!",0);

The numerical values at the end of the command arguments represent the state of the command and will send the text to a given log:

0 and 2 : ERROR and WARNING respectively, will send the text in the error log (log.e). Note that a WARNING (2) will not stop the running!

1 : INFOS, will send the text in the output log (log.o).

This function is highly complex, please do not modify it without the agreement of TOGGLE mainteners!

3.3.2 `toolbox::run`

This function will launch any command sent to it as argument (text) to the system, and will recover the exit status of the command. It will write the exact launched command in the output log, and any STDOUT also. All errors will be send to the error log and generally will drive to the stop of the pipeline.

To use this, respect the following nomenclature

```
toolbox::run(``my command to be launched'');
```

As for the previous function, `toolbox::run` is an intreasic function that cannot be modified except by mainteners.

3.4 The code itself

Let's come back to our example:

```

##SAMTOOLS SORT
#Sort alignments by leftmost coordinates.
sub samToolsSort{
    ...
    }
```

The `sub` is preceded by commentaries about the function and what it does

```

...
    my($bamFileIn,$optionsHachees)=@_;

...
```

input file and options are recovered through references.

```

    if (toolbox::sizeFile($bamFileIn)==1){ ##Check if entry file exist and is not empty

        #Check if the format is correct
        if (toolbox::checkSamOrBamFormat($bamFileIn)==0) {# The file is not a BAM/SAM file
            toolbox::exportLog("ERROR: samTools::samToolsSort : The file $bamFileIn is not a SAM/BAM file\n",0);
            return 0;
        }
        MY CORE COMMAND
    }else{
        toolbox::exportLog("ERROR: samTools::samToolsSort : The file $bamFileIn is uncorrect\n",0);
        return 0;#File not Ok
    }
```

We check if the input file exists (`toolbox::sizeFile`) and if the file is a SAM or a BAM (`toolbox::checkSamOrBamFormat`). If any error appears (empty file, wrong format), the script is stopped and logs filled using the `toolbox::exportLog function`.

```
        my $bamPath=toolbox::extractPath($bamFileIn);
        my $bamFileName=toolbox::extractName($bamFileIn).".SAMTOOLSSORT";
        my $bamFileOut = $bamPath.$bamFileName;
        my $options="";
        if ($optionsHachees) {
            $options=toolbox::extractOptions($optionsHachees);
        }
        my $command=$samtools." sort ".$options." ".$bamFileIn." ".$bamFileOut;
```

The `toolbox::extractPath` function allows to pickup file name without extension (similar to `basename` bash command). Using this shorter name, we can create an output name if required.

The `toolbox::extractOptions` function will create a text version of the hash containing the options for the given tool (first argument). A second optional argument can be provided to specify the separator between the option name and its value. Thus if the second argument is provided as "=", the option output would be "-d=1". Either, in standard it will be "-d 1" (standard is space).

The command line can thus be created.

```
        #Execute command
        if(toolbox::run($command)==1){
            toolbox::exportLog("INFOS: samTools::samToolsSort : Correctly done\n",1);
            return 1;#Command Ok
        }else{
```

```
6                       toolbox::exportLog("ERROR: samTools::
    samToolsSort : Uncorrectly done\n",0);
7                       return 0;#Command not Ok
8                   }
```

Once we have created the command, we can send it to `toolbox::run`, and report the output state (O, 1 or 2).

3.4.1 TIPS

Generally, the fastest and easiest way to create new functions is to copy an existing one (closely related) and to modify it.

3.5 The test

I have to fill this part....

Chapter 4

Creating a new block of code

Once the function is created, you can either stand like that, or adding it to the library of bricks we can use in the on the fly pipeline generation... Which is much greater :D

4.1 Already declared variables and other standard stuff

The on the fly version contains a wide range of already declared variables and standard way to know in which step we are. The already declared variables are (in the *startBlock.txt* file):

```
1  #Switch and populating variable
2
3 #Standard variables
4
5 my ($softParameters, $newDir,$stepOrder,$stepName,
     $fileWithoutExtention,$fastqForward,$fastqReverse,$samFile
     ,$extension,$samFileOut,$shortDirName,$stepF1,$vcfCalled,
     $vcfOut,$listOfBam,$bamFile,$bamFileOut,$cleanerCommand,
     $intervalsFile,$replacementCommand);
6
7 #Variable variables
8 my ($fastq1,$fastq2,$sam,$bam,$vcf);
9 my $fileList=toolbox::readDir($previousDir);
10 my $globalAnalysis=0; #to check if all data are to be treated
     one by one  (0) or in group (1)
```

In a same way, the current directory is already known, such as the previous one (see *previousBlock.txt* and *afterBlock.txt*).

4.2 Text block

A text block is an implementation of a call to the new function you designed. This code will be used latter by *toggleGenerator.pl* to generate the pipeline scripts. This is a text file, that must be saved in the *onTheFly* folder to be used.

Starting with the previous example, let's see what would be the code block associated with

```
1  ##########################################
2  # Block for samtools sort
3  ##########################################
4
5  $samFile=0;
6  foreach my $file (@{$fileList}) #Checking the type of files
       that must be SAM or BAM
7  {
8       if ($file =~ m/sam$|bam$/) # the file type is normally sam
        of bam
9       {
10           if ($samFile != 0) # Already a sam of a bam recognized
       , but more than one in the previous folder
11           {
12                toolbox::exportLog("ERROR : $0 : there are more
       than one single SAM/BAM file at $stepName step.\n",0);
13           }
14           $samFile = $file;
15      }
16 }
17
18 if ($samFile eq '0') #No SAM/BAM file found in the previous
       folder
19 {
20      toolbox::exportLog("ERROR : $0 : No SAM/BAM file found in
       $previousDir at step $stepName.\n",0);
21 }
22
23 $softParameters = toolbox::extractHashSoft($optionRef,
       $stepName);      # recovery of specific parameters of
       samtools sort
24
25 samTools::samToolsSort($samFile,$softParameters);    # Sending
       to samtools sort function
```

As you can see many controls and comments are added.

The first thing done is the checking of number of input file, as samToolsSort function will allow only one file to be sort at a time. It check also that the previous folder is not an empty one. Then it recovers the subhash containing the parameters for samtools sort, if any, and then send the arguments to the function.

As for creating a new function, the easiest way is to copy a related block, to modify it at conveniance then to save it under another name.

4.3 Indicating the input and output

The *softwareFormat.txt* file (root folder) allows the system to verify that the output of the step n are compatible with the input of step n+1.

It is basically informed in the following way:

```
$samToolsSort
IN=SAM,BAM
OUT=SAM,BAM
```

Multiple formats are separated by commas.

4.4 Providing the correct nomenclature

Last step, but not least, the adjustement of the nomenclature... In the code itself, we must respect the format previously described in this manual to call a given function.

However, the users are not du to respect this limitation in the *software.config* file. Thus, they can provide the function for our `samToolsSort` function using the correct nomenclature but also in different ways such as *samtools SORT* e.g.

The transformation/correction is ensured by the `namingConvention::correctName` function:

```
1  sub correctName
2  {
3      my ($name)=@_;
4      my $correctedName="NA";
5      my $order;
6      ## DEBUG toolbox::exportLog("+++++++++++++++$name\n",1);
7      my @list = split /\s/,$name;
8      $order = pop @list if ($list[-1] =~ m/^\d+/); # This is
       for a repetition of the same step
9      switch (1)
10     {
11     #FOR cleaner
12     case ($name =~ m/cleaner/i){$correctedName="cleaner";} #
       Correction for cleaner step
13
14     #FOR SGE
15     case ($name =~ m/sge/i){$correctedName="sge";} #Correction
        for sge configuration
16
17         #FOR bwa.pm
18         case ($name =~ m/bwa[\s|\.|\-|_\/|\\|\|]*aln/i){
       $correctedName="bwaAln"; } #Correction for bwaAln
19         case ($name =~ m/bwa[\s|\.|\-|_\/|\\|\|]*sampe/i){
       $correctedName="bwaSampe"} # Correction for bwaSampe
20         case ($name =~ m/bwa[\s|\.|\-|_\/|\\|\|]*samse/i){
       $correctedName="bwaSamse"} # Correction for bwaSamse
21         case ($name =~ m/bwa[\s|\.|\-|_\/|\\|\|]*index/i){
       $correctedName="bwaIndex"} # Correction for bwaIndex
22         case ($name =~ m/bwa[\s|\.|\-|_\/|\\|\|]*mem/i){
       $correctedName="bwaMem"} # Correction for bwaMem
```

```
23              ....
24          }
25  }
```

This function will recognize the names based on regular expression, and provide the correct name to the system. It remove spaces, points, dash, slash,... and recognize lower and upper case. To create your own entry, please use the following system

```
1   case ($name =~ m/my[\s|\.|\-|_\/|\\|\|]*name/i){
      $correctedName="myName"; } #Correction for myName function
```

Thus for samToolsSort the correction is:

```
1   case ($name =~ m/samtools[\s|\.|\-|_\/|\\|\|]*sort/i){
      $correctedName="samToolsSort"; } #Correction for
      samToolsSort function
```

As before, you can copy and modify a closely related line code.

4.5 Last but not least

Please commit all changes individually in YOUR branch and do not forget to push!

```
#Check your current status
git status

#Check the branch you are working on
git branch

#Perform the commit
git commit -m "My Explicite comment" changedFile

#Push your local branch to GitHub
git push https://github.com/SouthGreenPlatform/TOGGLE-DEV.git branchName
```

Appendix A

Licence