Extending flowQ: how to implement QA processes

F. Hahne

April 22, 2010

Abstract

flowQ provides infrastructure to generate interactive quality reports based on a unified HTML output. The software is readily extendable via modules, where each module comprises a single QA process. This Vignette is a brief tutorial how to create your own QA process modules.

1 Installation

The flowCore package can be installed using Bioconductor's `biocLite` function, making sure that all R dependencies are met. In addition, the ImageMagick library has to be installed on your system and its binaries have to be available in the system path. ImageMagick can be downloaded from `http://www.imagemagick.org`. On Debian linux systems, the command `sudo apt-get install imagemagick` usually works. A more detailed description of the ImageMagic installation procedure can be found in the *EBImageInstallHOWTO* Vignette of the EBImage package, which has similar dependencies.

2 Basic idea of flowQ's QA reports

In flowCore, flow cytometry data is organized in *flowFrames* and *flowSets*. Usually, a *flowSet* comprises multiple *flowFrames* for one complete experiment or the *flowFrames* of a single staining panel of one particular experiment. The initial step of all data analysis is typically a basic quality assessment (QA). Depending on the design of the experiment, the measurement channels and the biological question, there are various levels on which QA makes sense and also various different parameters which can be checked.

In flowQ we tried to implement a framework that allows to create consise QA reports for one or several *flowSets* and that is readily extendable using self-defined modules, thus allowing for arbitrary quality checks of the data. To this end, we define the abstract structure *QA process*. Essentially, a QA process can be viewed as a function that iterates over all *flowFrames* in a *flowSet* or even across several *flowSets*, assessing whether a particular check has been passed for the respective sample or set of samples, and that generates some sort of graphical output to indicate the findings. Some quality checks are independent of particular flow parameters, while others may check each parameter separately, in which case each of these checks constitutes a *QA subprocess*. The outcome for each subprocess can be indicated separately, however there also has to be an aggregated outcome for the whole QA process. This definiton is somewhat

abstract, and in the following we will present some more concrete examples which hopefully will make this structure much clearer.

The general design of a flowQ QA process is:

- aggregator: a qualitative or quantitative value that indicates the outcome of a QA process or of one of its subprocesses for one single *flowFrame* in the set.
- summary graph: a plot summarizing the result of the QA process for the whole *flowSet*.
- frame graphs: plots visualizing the outcome of a QA process or of one of its subprocesses for a single *flowFrame* (optional).

As mentioned before, a single QA process may contain various subprocesses, for instance looking at each measurement channel in a *flowFrame* separately. Each of these subprocesses may have its own aggregator and/or graphs. A unified aggregator indicating the overall outcome of the QA process is mandatory.

Abstractions for each of these building blocks are avaible as classes and for each class there are constructors which will do the dirty work behind the scenes. All that needs to be provided by the user-defined QA functions are file paths to the respective plots and lists of aggregators indicating the outcome (based on cutoff values that have been computed before. For each of these classes, there are `writeLine` methods, which create the appropriate HTML output. The user doesn't have to care about this step, a fully formated report will be generated when calling the `writeQAReport` function.

Figure 1 shows a schematic of a flowQ QA report.

3 Aggregators

There are several subclasses of aggregators, all inheriting from the virtual parent class *qaAggregator*, which defines a single slot, `passed`. This slot is the basic indicator whether the *flowFrame* has passes the particular quality check. More fine-grained output can be archived by the following types of sub-classes (see their documentation for details):

- *binaryAggregator*: the most basic aggregator, indicating "passed" or "not passed" by color coding. A green bulb indicates passing of the check and a red bulb indicates failing.

- *discreteAggregator*: allows three different states: "passed", "not passes" and "warn", also coded by colored bulbs. Warnings are indicated by yellow color. Note that "warn" will set the `passed` slot to `FALSE`.

- *factorAggregator*: multiple outcome states. The factor levels are plotted in gray along with the actual factor value, color coded for the overall outcome ("passed" in green or "not passed" in red).

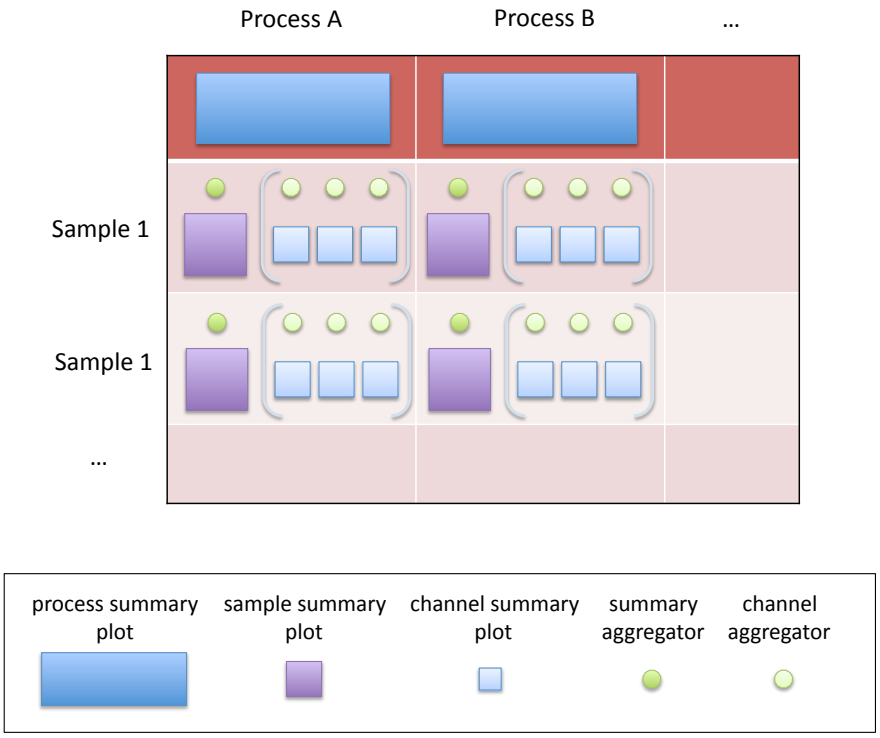

Figure 1: Schematics of an flowQ QA report. Columns in the table represent different QA criteria, rows represent samples. For each criterion there may be a single overall summary plot. For each sample in the respective QA criterion there may be a single detailed plot as well as several channel-specific plots if necessary. Accordingly, for each sample, the overall QA result is sumamrized by a single aggregator, and optional channel-specific results can be summarized by additional channel aggregators.

- *stringAggregator*: arbitrary character string describing the outcome. Font color indicates the overall outcome ("passed" in green or "not passed" in red).

  passed
  failed

- *numericAggregator*: a numerical value describing the outcome. Currently, the value is plotted as a character string, but this might change in the future. Font color indicates the overall outcome ("passed" in green or "not passed" in red).

  1 0

- *rangeAggregator*: a numerical value within a certain range describing the outcome. A horizontal barplot is produced with color indicating the overall outcome ("passed" in green or "not passed" in red).

Aggregator objects can be created using their constructor functions of the same name as the respective class. The input to the constructors varies depending on the type of the aggregator, e.g. the `binaryAggregator` function takes a logical scalar. See their documentation for further details. The following code creates instances of each of the six aggregator types:

```
> binaryAggregator()

Binary quality score passing the requirements

> discreteAggregator(2)

Discrete quality score not passing the requirements with state warn

> factorAggregator(factor("a", levels = letters[1:3]))

Factorized quality score passing the requirements of value=a

> stringAggregator("test", passed = FALSE)

Textual quality score not passing the requirements of value=test

> numericAggregator(20)

Numeric quality score passing the requirements of value=20

> rangeAggregator(10, 0, 100)

Range quality score passing the requirements of value=10
```

A special class *aggregatorList* exists that holds multiple aggregators, not necessarily of the same type, and this is used for QA processes with several subprocesses. The constructor takes an arbitrary number of *qaAggregator* objects, or a list of such objects. This class mainly exists for method dispatch.

```
> aggregatorList(bin = binaryAggregator(FALSE), disc = discreteAggregator(1))

List of 2 aggregators
```

4 Storing images as *qaGraph*s

While aggregators indicate the general outcome of a QA process, or, at most, a single quantitative value, the amount of information they can provide is very limited. flowQ's design allows to include additional diagnostic plots, both on the level of the whole *flowSet* and for each *flowFrame* individually. Smaller bitmap versions of the plots are used for the overview page, and each image is clickable, opening a bigger vectorized version of the plot that is better suited for detailed inspection. This feature can be turned off in order to minimize the size of the final report, since pdf versions of the plots tend to be much bigger than the bitmap versions. To take the burden of file naming and file conversion away from the user, the class *qaGraph* was implemented, which stores links to single images and whose initializer either creates vectorized versions from bitmaps or vice versa. The class constructor takes two mandatory arguments: `fileName` which is a valid path to an image file (either bitmap or vectorized), and `imageDir`, which is a file path to the output directory where the image files are to be stored. If you are planning to place the final QA report on a web server, you should make sure, that this path is accessable. The safest solution is to chose a directory below the root directory of the QA report, e.g., `qaReport/images` if the root directory is `qaReport`.

You can control the final width of the bitmap version of the image through the optional `width` argument, and empty *qaGraph* objects can be created by setting `empty=TRUE`. During object instantiation, the file type is detected automatically and the image file will be converted, resized and copied if necessary.

```
> tmp <- tempdir()
> fn <- file.path(tmp, "test.jpg")
> jpeg(file = fn)
> plot(1:3)
> dev.off()

pdf
  2

> idir <- file.path(tmp, "images")
> g <- qaGraph(fn, imageDir = idir)
> g

QA process image information

> qaGraph(imageDir = idir, empty = TRUE)

QA process image information
```

For the special case of QA processes with multiple subprocesses (e.g., individual plots for each channel), there is a class *qaGraphList* and an associated constructor, which will take a character vector of multiple file names. This class mainly exists for method dispatch and to facilitate batch processing of multiple image files.

5 Information for a single frame: class *qaProcessFrame*

All the information of a QA process for a single frame has to be bundled in objects of class *qaProcessFrame*. Again, a constructor facilitates instantiating these objects; the mandatory arguments of the constructor are:

- `frameID`: a unique identifier for the *flowFrame*. Most of the time, this will be the `sampleName` of the frame in the *flowSet*. The frame will be identified by this symbol in all of the following steps and you should make sure that you use unique values, otherwise the downstream functions will not work.
- `summaryAggregator`: an object inheriting from class *qaAggregator* indicating the overall outcome of the process for this frame.

Further optional arguments are:

- `summaryGraph`: an object of class *qaGraph* providing a graphical summary of the whole QA process for this frame. Unless there are several subprocesses, this is the natural choice for a single plot depicting the outcome for a particular frame, however this infrastructure allows all possible combinations (no overall plot but plots for each suprocess, only plots for the subprocesses, no plots at all).
- `frameAggregators`: an object of class *aggregatorList*. Each aggregator in the list indicates the outcome of one single subprocess for this frame, e.g., for every individual measurement channel.
- `frameGraphs`: an object of class *qaGraphList*. Each qaGraph in the list is a graphical overview over the outcome of one single subprocess for this frame. Note that the length of both `frameAggregators` and `frameGraphs` have to be the same if you want to provide graphical output for subprocesses. Assuming that you don't want to include images for one or several of the subprocesses, you have to provide empty *qaGraph* objects (see above). It is not possible to omit aggregators for subprocesses, because they are used to link to the repsective images.
- `details`: a list of additional information that you want to keep attached to the *qaProcessFrame*. For example, this can be the values of a quality score that was computed in order to decide whether the QA process has passed the requirements. Such information can be useful to update aggregators later without reproducing the images (e.g., when a cutoff value has been changed). There is no fixed structure for this list, and it is up to the developer to provide useful information as well as and update function which utilizes this information and changes the aggregator outcome status.

6 The whole QA process: class *qaProcess*

Now that we have all the information for the single frames together, we can proceed and bundle things up in a unified object of class *qaProcess*. Again, the constructor has a couple of mandatory arguments:

- `id`: a unique identifier for this QA process. This will be used to identify the process in all downstream functions, which will not work unless it really is unique (assuming that you want to combine multiple QA processes in one single report).
- `type`: A character scalar describing the type of the QA process. This might become useful for functions that operate on objects of class *qaProcess* in a type-specific way (e.g., updating agregators).
- `frameProcesses`: a list of *qaProcessFrame* objects. You have to make sure that the identifier for each *qaProcessFrame* is unique and that the length of the list is equal to the length of the *flowSet*. We strongly recommend taking the `sampleNames` of the *flowSet* as unique identifiers for the individual *qaProcessFrames* to avoid mix-ups.

Further optional arguments are:

- `name`: The name of the process that is used as caption in the output. If this is not provided, the constructor tries to come up with a more or less intelligent default, and it is always better to explicitely name the process.
- `summaryGraph`: An object of class *qaGraph* summarizing the outcome of the QA process for the whole frame. Although this is optional, we strongly recommend including such a plot, as it provides a good initial overview.

The output of you own QA process function should always be an object of class *qaProcess*, which can be used in the downstream functions to produce the quality assessment report. Most of the time, the function would have a structure similar to the following:

1. iterate over frames to create the *qaProcessFrame* object, possibly with an additional level of iteration for each subprocess (e.g. each channel). Each iteration involves creation of (usually at least one) *qaGraph* and at least one *qaAggregator* object.
2. create an *qaProcessFrame* object summarizing the process for the whole set.
3. bundle things up in a *qaProcess* object.

7 Examples

In this final section we are going to provide a couple of very simple examples for potential QA process functions in order to get you started. Note that the inputs to such a function are totally free, and you could build arbitrarily complex procedures, potentially comparing features across several *flowSets*, groups of *flowFrames* or any other checks you deem to be valuable. However, the output always has to be a single object of class *QAProcess*.

7.1 Simple process without subprocesses

We start with an very simple example which is — in a slightly more sophisticated version — already a part of the package in the function `qaProcess.cellnumber`. The idea is to

take a single *flowSet* as input and check whether the total number of cells for the individual *flowFrames* is above a certain threshold. The user should be able to specify the *flowSet*, the threshold value, the output directory in which to generate the images and the name of the process which will later appear in the report. We could use the following function header:

```
> cellnumber <- function(set, threshold = 5000, outdir,
+     name = "cellnumber") {
+ }
```

We first have to think about reasonable graphical outputs for our report. A barplot seems to be a good idea to show total cell numbers for each sample in the summary graph. The check is trivial for individual sample, and it doesn't make much sense to provide plots for each of them, after all we are simply checking whether a number is bigger or smaller than another number. This reasoning also suggests a aggregator type to use: the *numericAggregator*. Let's start with the overview plot.

```
> cellnumber <- function(set, threshold = 5000, outdir,
+     name = "cellnumber") {
+     if (!file.exists(outdir))
+         dir.create(outdir, recursive = TRUE)
+     cellNumbers <- as.numeric(fsApply(set, nrow))
+     sfile <- file.path(outdir, "summary.pdf")
+     pdf(file = sfile)
+     col <- "gray"
+     par(mar = c(10.1, 4.1, 4.1, 2.1), las = 2)
+     barplot(cellNumbers, col = col, border = NA, names.arg = sampleNames(set),
+         cex.names = 0.8, cex.axis = 0.8)
+     abline(h = mean(cellNumbers), lty = 3, lwd = 2)
+     dev.off()
+     sgraph <- qaGraph(fileName = sfile, imageDir = outdir)
+ }
```

In the above code we are making sure that the output directory is created if necessary, record the number of cells for each *flowFrame* in the *flowSet* and produce a barplot in the pdf file `summary.pdf`. In a subsequent step, we create a qaGraph object from this filename and we also specify to use the output directory to store the images in. We could have created the initial pdf file anywhere on our system, for instance in a temporary folder. The constructor is smart enough to copy it into the output directory if necessary, and it will also create this directory if it doesn't exist.

This already looks nice, but we are not done yet. Now it is time to create the aggregators for our report. We have to supply them to the qaProcess constructor in the form of a list, so we first create a list of appropriate length. We then iterate over each *flowFrame* in order to check whether the number of cells is above the threshold and create the appropriate *numericAggregator* objects. In a final step we provide both the *qaGraph* object containing the summary plot as well as the `frameProcesses` list and the user-defined name for the process to the `qaProcess` constructor function.

```
> cellnumber <- function(set, threshold = 5000, outdir,
+     name = "cellnumber") {
+     if (!file.exists(outdir))
+         dir.create(outdir, recursive = TRUE)
+     cellNumbers <- as.numeric(fsApply(set, nrow))
+     sfile <- file.path(outdir, "summary.pdf")
+     pdf(file = sfile)
+     col <- "gray"
+     par(mar = c(10.1, 4.1, 4.1, 2.1), las = 2)
+     barplot(cellNumbers, col = col, border = NA, names.arg = sampleNames(set),
+         cex.names = 0.8, cex.axis = 0.8)
+     dev.off()
+     sgraph <- qaGraph(fileName = sfile, imageDir = outdir)
+     frameIDs <- sampleNames(set)
+     frameProcesses <- vector(mode = "list", length = length(frameIDs))
+     for (i in seq_along(frameIDs)) {
+         agg <- new("numericAggregator", x = cellNumbers[i],
+             passed = cellNumbers[i] > threshold)
+         frameProcesses[[i]] <- qaProcessFrame(frameIDs[i],
+             agg)
+     }
+     return(qaProcess(id = "cellnumprocess", name = name,
+         type = "cell number", summaryGraph = sgraph,
+         frameProcesses = frameProcesses))
+ }
```

As you can see, the function is not particularly complicated, and the biggest part was actually creating the barplot. We didn't have to worry at all about the structure of the final report or any of the details of displaying the provided information. This is the idea behind the design of this package. The only things you should worry about are:

1. Which particular aspect of the data do I want to check?
2. How (if at all) can I visualize this aspect for the whole *flowSet*?
3. How (if at all) can I visualize this aspect for each *flowFrame*?
4. What is my "test statistic"? Is it a simple yes now answer? Is is a quantitative value? Which type of aggregator best fits the test?

7.2 Subprocesses

This next example is a little bit more complicated in the sense that we want to check each of the flow parameters in the *flowSet* separately. The question is how many events are squished on the borders of the measurement range for each parameter. Again, a similar but slightly more complex function is implemented as part of the package (`qaProcess.marginevents`). As

explained earlier, the separate checks on each parameter constitute several subprocess, and consequently we have one additional level of complexity here.

We start again by first defining the inputs to our function. In addition to the threshold, which should be a percentage between 0 and 100, we want to give the user the choice of which flow parameters to include in the analysis. The default is to do it for all available parameters.

```
> marginevents <- function(set, threshold = 10, channels = colnames(set),
+     outdir, name = "margin events") {
+ }
```

Next we have to find out how many margin events there are for each frame and for each parameter. Conveniently, the *boundaryFilter* defined in flowCore does exactly that. We can wrap this in another function.

```
> mevents <- function(set, channels) {
+     sapply(channels, function(x) {
+         ff <- filter(set, boundaryFilter(x))
+         sapply(ff, function(y) summary(y)$p)
+     })
+ }
```

Now it is again time to decide about informative visualizations. For the overview plot over all samples and parameters, and image plot would be quite helpful. We will use the levelplot function defined in the lattice package for this purpose. It might also be nice to get a better understanding of why there are a lot of margin events for a particular parameter and sample. This could for instance be caused by a shift of the underlying data. We should be able to see such effects in density plots of the data. If we try to translate this back into the abstract flowQ framework, we realize that we will need one single *qaGraph* object for each subprocess within each frame. We don't really have a need for a per-frame summary graph, so we can skip this step. There might be other situations where it makes sense to have both the parameter-specific plots, frame-specific summary plots and also overall summary plots for the whole *flowSet*. We will first take care of the summary plots.

```
> marginevents <- function(set, threshold = 10, channels = colnames(set),
+     outdir, name = "margin events") {
+     perc <- mevents(set, channels)
+     require("lattice")
+     tmp <- tempdir()
+     sfile <- file.path(tmp, "summary.pdf")
+     pdf(file = sfile)
+     col.regions = colorRampPalette(c("white", "darkblue"))(256)
+     print(levelplot(t(perc) * 100, scales = list(x = list(rot = 90)),
+         xlab = "", ylab = "", main = "% margin events",
+         col.regions = col.regions))
+     dev.off()
+     sgraph <- qaGraph(fileName = sfile, imageDir = outdir)
+ }
```

In order to check whether any of the margin event counts is above the threshold and to produce the density plots we have to iterate over both frames and parameters. The constructor for the *qaFrameProcess* object needs a little more input this time. As explained before, we can supply the graphs for each of the subprocesses by means of a *qaGraphList*, which again can be created from a character vector of file names using the `qaGraphList` constructor. So one thing we need to keep track of while iterating over parameters are the file names. For each parameter in each frame we also want to create an appropriate aggregator. We said this before: for each subprocess we need a separate aggregator. As a matter of fact, we tell the software about the presence of subprocesses by supplying multiple aggregators for a frame. Our test statistic is a percentage, so we know that all values fall in the range between 0 and 100. The *rangeAggregator* seems to be a good choice here, since it won't only tell us whether the check has passed or failed, but even indicate the magnitude of the effect. The individual aggregators need to be stored in some form of a list, and the *aggregatorList* class is the appropriate container. You can think of this class as a regular list, and it mainly exists for internal software design purpose. We fill it in exactly the same way as we would fill a regular R list.

To complete the *qaProcess* object we also need a summary aggregator. In this case, we came up with a simple rule: if one of the subprocesses doesn't pass we want to issue a warning. If more than two fail, a failed overall quality check should be indicated and if all of the subprocesses pass the test we are happy to report no problems at all. The final call to the `qaProcess` constructur doesn't really look any different than in our previous example. Indeed, all the additional complexity is encapsulated in the *qaProcessFrame* objects.

```
> marginevents <- function(set, threshold = 10, channels = colnames(set),
+     outdir, name = "margin events") {
+     perc <- mevents(set, channels)
+     require("lattice")
+     tmp <- tempdir()
+     sfile <- file.path(tmp, "summary.pdf")
+     pdf(file = sfile)
+     col.regions = colorRampPalette(c("white", "darkblue"))(256)
+     print(levelplot(perc * 100, scales = list(x = list(rot = 90)),
+         xlab = "", ylab = "", main = "% margin events",
+         col.regions = col.regions))
+     dev.off()
+     sgraph <- qaGraph(fileName = sfile, imageDir = outdir)
+     frameIDs <- sampleNames(set)
+     frameProcesses <- list()
+     for (i in 1:length(set)) {
+         fnames <- NULL
+         agTmp <- aggregatorList()
+         for (j in 1:length(channels)) {
+             tfile <- file.path(tmp, paste("frame_", sprintf("%0.2d",
+                 i), "_", gsub("\\..*$", "", channels[j]),
+                 ".pdf", sep = ""))
+             pdf(file = tfile, height = 3)
```

```
+             par(mar = c(1, 0, 1, 0))
+             plot(density(exprs(set[[i]][, channels[j]])),
+                 main = NULL)
+             dev.off()
+             fnames <- c(fnames, tfile)
+             passed <- perc[i, j] < threshold/100
+             agTmp[[j]] <- new("rangeAggregator", passed = passed,
+                 x = perc[i, j], min = 0, max = 1)
+         }
+         names(agTmp) <- channels
+         nfail <- !sapply(agTmp, slot, "passed")
+         val <- if (sum(nfail) == 1)
+             factor(2)
+         else if (sum(nfail) == 0)
+             factor(1)
+         else factor(0)
+         ba <- new("discreteAggregator", x = val)
+         fGraphs <- qaGraphList(imageFiles = fnames, imageDir = outdir)
+         frameProcesses[[frameIDs[i]]] <- qaProcessFrame(frameID = frameIDs[i],
+             summaryAggregator = ba, frameAggregators = agTmp,
+             frameGraphs = fGraphs)
+     }
+     return(qaProcess(id = "processmarginevents", name = name,
+         type = "margin events", summaryGraph = sgraph,
+         frameProcesses = frameProcesses))
+ }
```

Rather than specifiying a strict cutoff for a quality assessment process, it is often more reasonable to try to find outlier samples based on a certain test statistic in comparison to all other samples or all other samples in the same sample group. The *qaProcesses* implemented in `qaProcess.cellnumber` and `qaProcess.marginevents` use such dynamic assessment, but other than that they are pretty similar to the above examples.

8 Further extensions of the framework

It is worth mentioning here that the flowQ's framework can be extended even further. Since the individual building blocks like aggregators are all abstract classes, one could easily add more complicated ones. For the purpose of producing the interactive output via `writeQAReport`, all that is needed in addition to the extended aggregator class definition is a `writeLines` method which generates the necessary HTML code.

flowQ is also not limitted to producing fancy HTML output. The qaProcess object holds all the necessary information, and it is straight forward to extract this in order to put it into a data base, or to simply create a table containing the QA results. The details of such extensions are beyond the scope of this document, but the interested reader is referred to the documentation of the classes presented before, in particular that of *qaProcess* and *qaProcessFrame*.