lusis

TANGO – Development
Practical Exercises

REFERENCE	: Practical exercises	Version	: 02.01	Date	: 2020-01-06
VISA	:	Status	: Reviewed	Number of pages	: 60

Distribution

Authors	Paul Ng	Lusis
	Marc Foiret	Lusis
	Hugues Lefebvre	Lusis
	Lionel Mangin	Lusis
	Pierre Morin	Lusis

Reviewers	Michael Valdenaire	Lusis
	Hugues Lefebvre	Lusis

Recipients	Michael Valdenaire	Lusis
	Laurent Laperrousaz	Lusis

Revision history

Version	Date	Description
01.00*	2016-02-05	Initial draft - Add unit test - Add process ITGS (ITG server), DISPG - ISSUER service moved to a dedicated ISSUER process - transaction context has been renamed request context (mcp_ctx) - Exercise 3 (send sub-request) dispatcher error exercise reviewed - Exercise 9 (consignation) moved to exercise 4 Added extension manageCommitRollback() here - Rework on file logging to take the new tags into account: VERBOSITY, DATAFORMAT, DATADESC and COMMENT - Exercise 11 (Web Service) reworked based on last TANGO components tg_wsruntime and tg_wsgenerator - tg_batch logic explained - Exercise 12 (TG_BATCH) renamed into ATM simulator
01.01	2016-04-01	- Paragraph 9.2, 18.2, 19.2, 20.2, 21.2: typo `regen_prod.sh` instead of `regen_prod.pl` - Timer `setRefCli` added
02.00	2019-12-09	- Update for Tango V7 - Add diagrams for TG_CLIAPP event processing algorithms (paragraph 3) - Add connection procedure with Jupyter and Gitea - Add database connection information (paragraph 10.2) - Review TG_CLIAPP command API (paragraph 16) - Exchange exercise 11 and 12
02.01	2020-01-06	- Delete exercise 12: Web services - Add part 1 to 4 to group relevant chapters together - Add appendix (part 4) for advanced or optional extensions

* Note: this document a complete review of Practical exercises documents described in references [1] and [2].

Components version

See LSC **Tango Training 2.1.0** for the components version
https://lsc.lusis.net/lsc-support/index.php?r=product/view&id=214

Table of contents

List of figures

List of tables

Empty section

References

Ref.	Title
[1]	cnce-tc-00-07-V01.03.doc (Legacy French version of practical exercises)
[2]	Practical exercises (English version, undocumented version)
[3]	tango-dev-hand-book-2-9.doc
[4]	
[5]	
[6]	
[7]	
[8]	
[9]	
[10]	

Writing conventions

Important notions of Tango are written in **bold characters**.

Piece of code, TG_CLIAPP methods and UNIX commands are written with the `Courier New` font.

Overlodable TG_CLIAPP methods are written with the `Courier New` font in blue.

The Tango bus fields are written in italic. Example for transaction amount: *transactionAmount*

1. INTRODUCTION

1.1. Audience and prerequisites

This document is intended for developers who are going to work with the Tango Framework. It assumes that the developer has previous knowledge about the following:

- C++
- Basic UNIX commands and bash scripting
- MySQL

Also, it is assumed that the developers have completed the courses concerning the Tango principles and Tango environment.

1.2. Contents of the document

This document is divided into four sections:

Section 1 (paragraphs 2 and 3) presents the basic concepts of the TG_CLIAPP base class. This class is the basis of nearly all Tango applicative class.

Section 2 from paragraph 4 to 6 describes the general setup and compilation procedure.

Section 3 from paragraph 7 to 25 describes the differents exercises to do, along with explanations of the concepts.

Section 4 from paragraph 26 to the end of the document contains annexes.

PART 1: General concepts of TG_CLIAPP

2. TG_CLIAPP

2.1. Presentation

TG_CLIAPP class is a C++ class that hides the Tango system classes so that the application developer can concentrate on writing algorithmic functions without any constraints on system considerations.

TG_CLIAPP class consists of methods that are called automatically during different life stages of the Tango application, e.g. instance initialization, instance configuration update, request and answer reception and so on.

2.2. Class diagram

The class diagram is provided below for information. Knowledge of parent TG_CLI, TG_ROOT classes are not required to do the exercises.

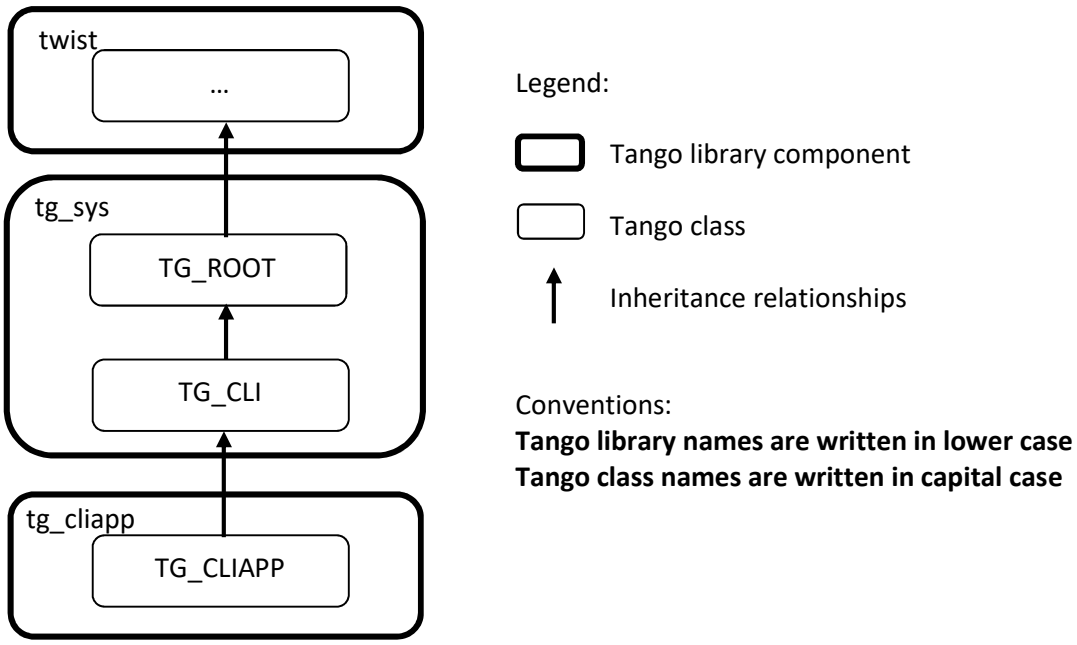

Figure 1: TG_CLIAPP class diagram

TG_CLIAPP is not usable as is, one has to create an applicative class that inheritates from TG_CLIAPP.

2.3. TG_CLIAPP in Tango services

TG_CLIAPP is omnipresent in Tango services.

In the Tango environment configuration, the modules dependencies can be seen in the config/modules.xml file.

In the example below, JNL service of JNL01 process executes the EMA_LOGMGR class. Looking at the config/modules.xml file, EMA_LOGMGR depends on EMA_LOGGEN, which in turn depends on TG_CLIAPP.

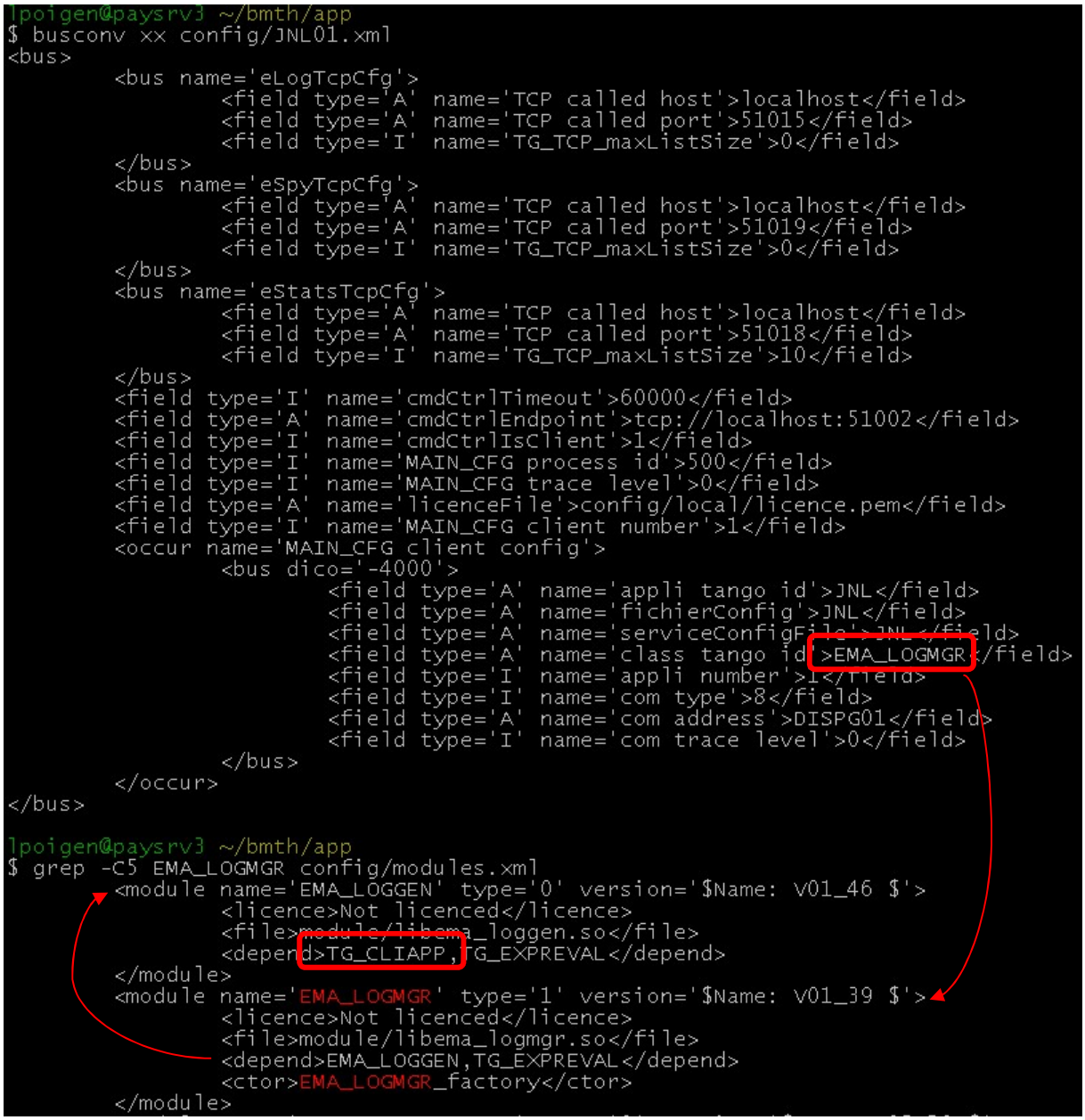

Figure 2: TG_CLIAPP in config/modules.xml

In a running Tango environment, this can be seen in the different process .out files in the log/ subfolder of the Tango application. From the previous example, the JNL process .out file would display the class dependencies.

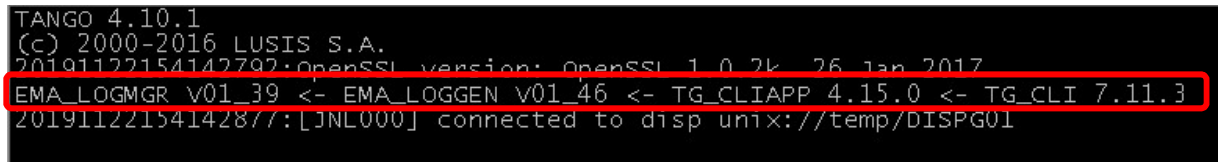

Figure 3: Another view of TG_CLIAPP dependencies

3. CHARACTERICTICS OF TG_CLIAPP

3.1. Accessors

TG_CLIAPP provides accessors that are pointers to classes for the manipulation of external data used by the application instance. Manipulating those fields does not require the application developer to call backend API so it abstracts access to these data from real implementation, and if the way data are stored in the framework changes, there is no need to modify application code.

3.1.1. The autobus: mcp_bus

The **autobus** is an accessor that simplifies access to TANGO bus, it is loaded:

- When application receives a request
- When application receives a response
- When application receives a notification

The autobus provides methods allowing to:

- Read the value contained by one of the bus fields
- Modify field values or add new fields
- Delete fields
- Check fields presence

The autobus is unique and is instantiated once in the applicative service. It is initialized with the incoming request, response or notification bus message. It is also used by default to send notifications or sub-requests.

The autobus is **NOT** a Tango bus; **it is loaded from the real Tango bus on incoming event** and can be devirtualized into a real Tango bus. **To put it simply, the autobus acts as a container which content is reset on each incoming request/response/notification. There is no persistency of data in the autobus.**

3.1.2. The request context: mcp_ctx

TG_CLIAPP provides a way for programmers to store data after sending a request and before receiving its response: **the request context**.

A unique number is generated to pair the response from the request in TG_CLIAPP. This number is accessible via the ***refSess*** field of mcp_ctx.

The request context is partly managed by TG_CLIAPP:

- A new request context and its refSess number are automatically created when TG_CLIAPP receives a request
- The request context is destroyed automatically by TG_CLIAPP when the response has been sent
- If a sub-request is sent, then the sub-response will be automatically paired with the corresponding sub-request based on the refSess

Application developers only need to store data that should be persistent between the request and the response. Request context provides methods allowing to:

- Write data in the request context
- Read data from the request context

Note: It is not possible to delete a value from a request context.

For other incoming events, such as notification, command, and so on, the request context is not defined. In this case, it must be handled by the application developer with methods that:

- Manually creates and destroys request contexts
- Manually finds transaction context

3.1.3. The session context: mcp_ses

It can be useful to store some data from one request to another, in this purpose TG_CLIAPP provides a session context mechanism.

It works in the same way as request contexts but allocating, destroying and finding session contexts is in charge of the application developer, there is no automatic handling.
Session context provides methods allowing to:

- Create a session context
- Write data in the session context
- Read data from the session context
- Find one session context using multiple keys
- Destroy a session context

The developer must ensure:

- That every unused session context will be destroyed
- That the maximum number of simultaneous allocated context is limited and reasonable
- That there is, at least, one data that can be used as a unique key to search for the right session

3.1.4. The configuration: mcp_cfg

Applications get their configuration data through the TANGO API. These can be system or application data.
TG_CLIAPP provides read accessor to the configuration data. The configuration is read at startup of the application service or the reception of a command requesting the configuration reloading.

3.2. General algorithms of TG_CLIAPP

3.2.1. Service initialization and start

On startup, TG_CLIAPP follows the diagram below to initialize the SQL database connection and queries and the configuration parameters:

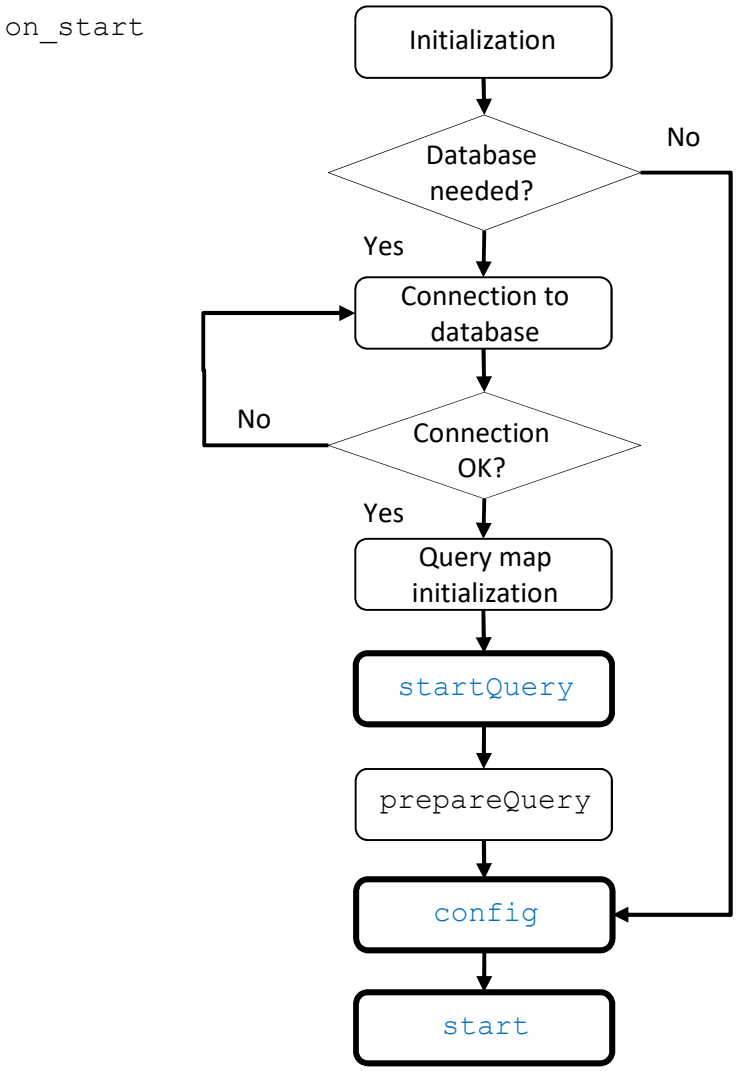

Figure 4: TG_CLIAPP on_start processing

3.2.2. Event processing

After the service has started, TG_CLIAPP waits for incoming events to be processed, as shown in the diagram below. When an event arrives, the `mainExcept` method is called. The main purpose of `mainExcept` is to catch the C++ exceptions (with try/catch) that may occur in the `on_xxx` processing methods and finish properly with `endRequest`.

The `endRequest` method is called afterwards to determine whether a response or a command reply should be issued or not.

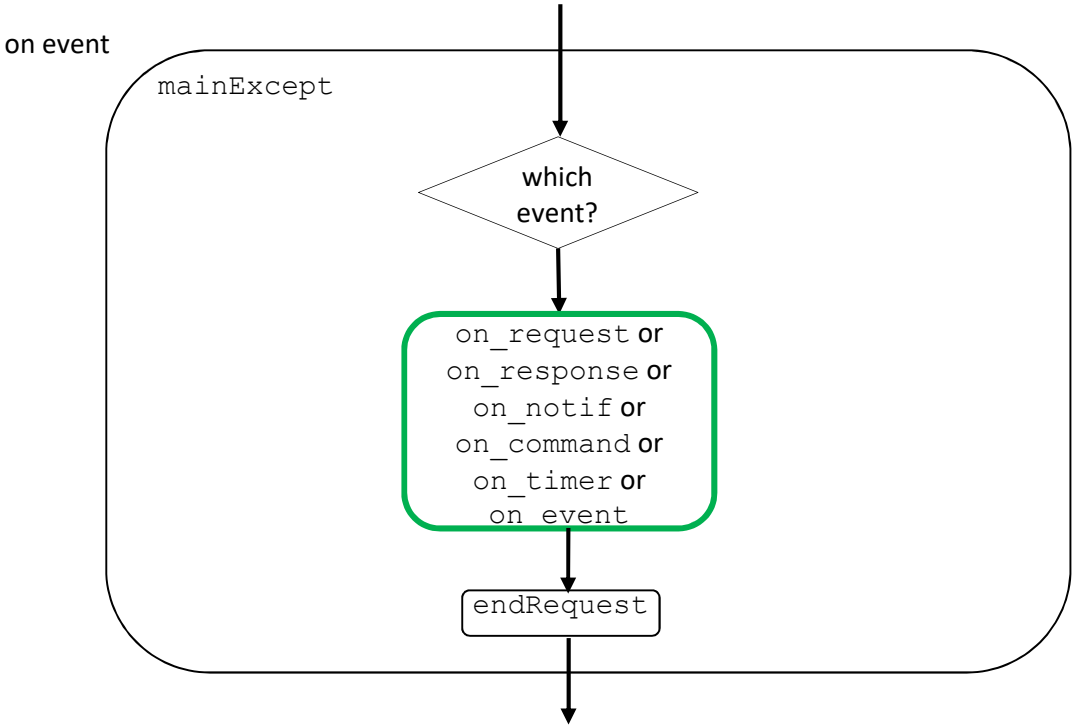

Figure 5: TG_CLIAPP on event processing

3.2.2.1. Internal code

`m4_codeInterne` is a member data of TG_CLIAPP which the developer should set to a value indicating the processing result. By default, `m4_codeInterne` is set to `0` (undefined).

The accepted values of `m4_codeInterne` are:

- `k4_a1` to `k4_a5`: nominal processing values. Only `k4_a1` is used in current implementation code
- `k4_i0`: pending value (do not respond)
- `k4_r1` to `k4_r7`: TANGO technical error. Do not use these codes as they are set by TANGO framework.

- `k4_x1` to `k4_x29`: TANGO applicative error. Developers should set this internal error codes.

These constants are all defined at framework level and are available so there is no need to define them in applicative code.

3.2.2.2. `on_request` processing

TG_CLIAPP provides methods that the developer can overload to handle incoming requests.

TG_CLIAPP allows the developer to split the incoming requests in two groups, the "direct requests" and the "reverse requests". This division is virtual and on the initiative of the developer. TG_CLIAPP calls a method called `whichDirectionRequest()`, that will address direct requests by default if it is not overloaded by the developer.

When the developer wants to handle "direct requests", one has to overload the `algoRequest()` method and, for "reverse requests", it the `algoRequestRevers()` method.

Upon reception of a request by applicative instance, TG_CLIAPP will:

- Create a request context
- Load the incoming bus into the autobus
- Set the internalCode to 0
- Determine the kind of request by calling `whichDirectionRequest()`
- Call the method which is designed to handle this kind of request and which will perform algorithmic processing.

`on_request`

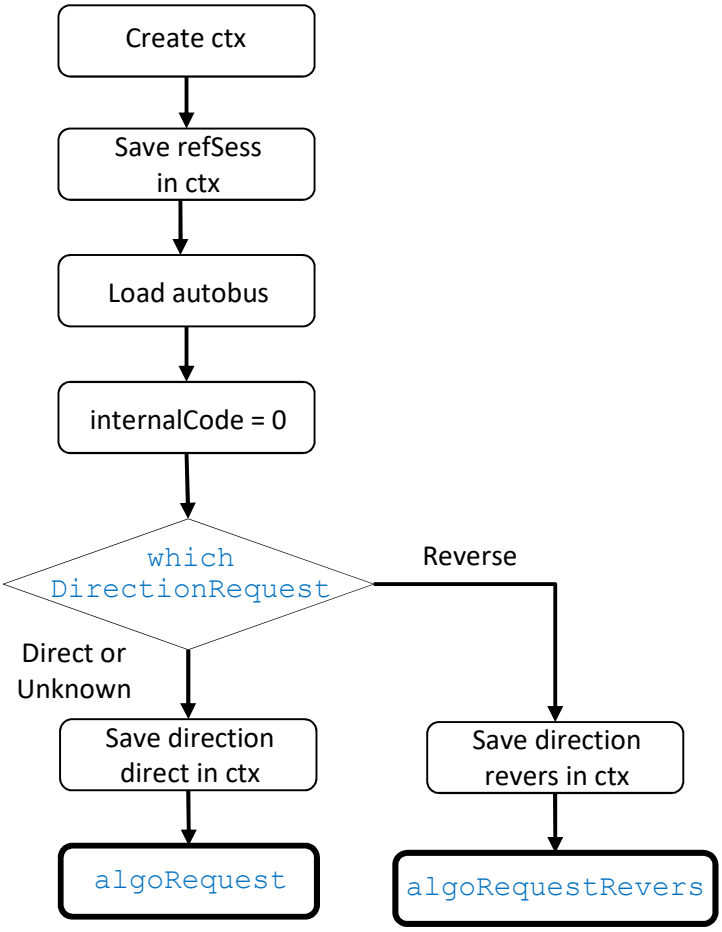

Figure 6: `on_request` processing

When `algoRequestXXX()` function returns, TG_CLIAPP will proceed with sending the answer unless the internalCode was set to the constant value **`k4_i0`**.

Blocking the automatic sending of the answer supposes that we will send the answer later. If we don't, the dispatcher will send a failure to issuer with timeout errorCode.

In most cases, the blocking of the automatic sending of the answer is used in order to send one or several sub-requests and wait for their answers before responding to the main request.

3.2.2.3. `on_response` processing

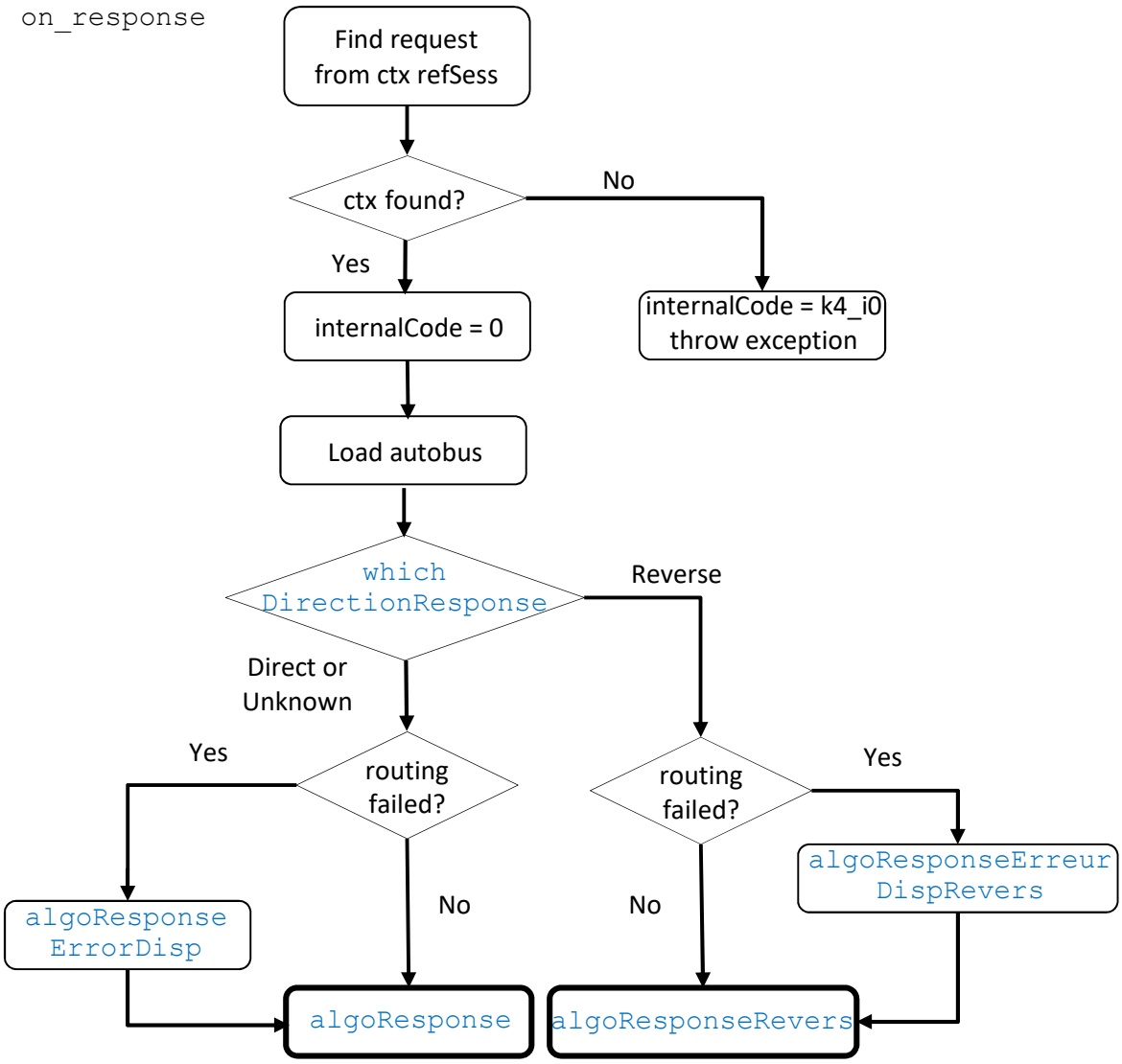

Figure 7: `on_response` processing

When a response is received by TG_CLIAPP, it will first try to look for the matching request based on the ***refSess***. If the matching fails, then an internal exception is raised and the response is trashed silently. In particular, this happens when a late response arrives after the dispatcher times out.

Responses come from two sources, either:

- the recipient of the request, in which case `algoResponse` or `algoResponseRevers` is called
- the dispatcher (because of a routing failure or timeout), in which case the overloadable methods `algoResponseErreurDispXXX` are called before `algoResponseXXX`.

When `algoResponseXXX()` function returns, TG_CLIAPP will proceed with sending the answer unless the internalCode was set to the constant value **`k4_i0`**.

3.2.2.4. `on_notif` processing

Processing of incoming notification is similar to request, except that there is no context request management in this case.

`on_notif`

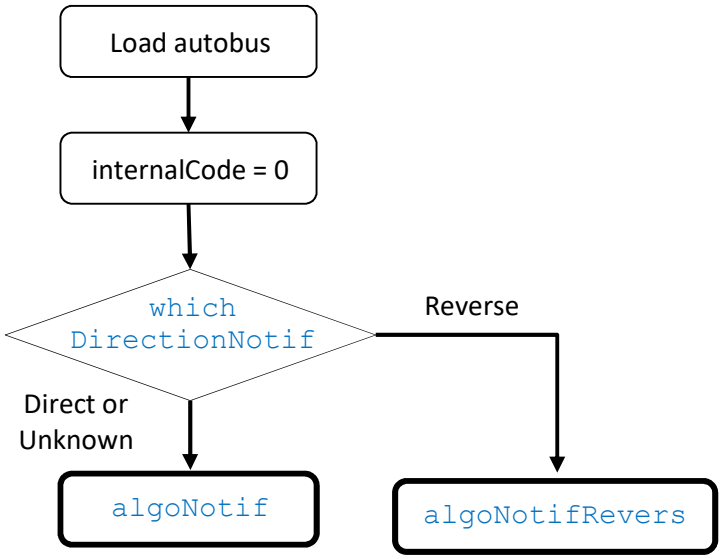

Figure 8: `on_notif` processing

3.2.2.5. `on_command` processing

See paragraph 17.2 Command processing

3.2.2.6. `on_timer` processing

See paragraph 19.4 Timer processing

3.2.2.7. `on_event` processing

This is used to process other types of events by TG_CLIAPP. It is unlikely to be seen.

`on_event`

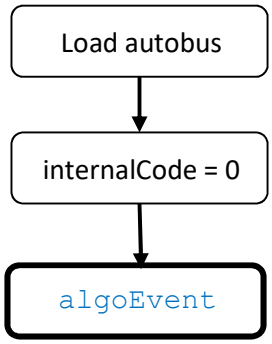

Figure 9: `on_event` processing

3.2.2.8. `endRequest` processing

At the end of event processing, TG_CLIAPP assesses the need to respond to the request or command. Below is the activity diagram that resumes the steps executed by `endRequest`.

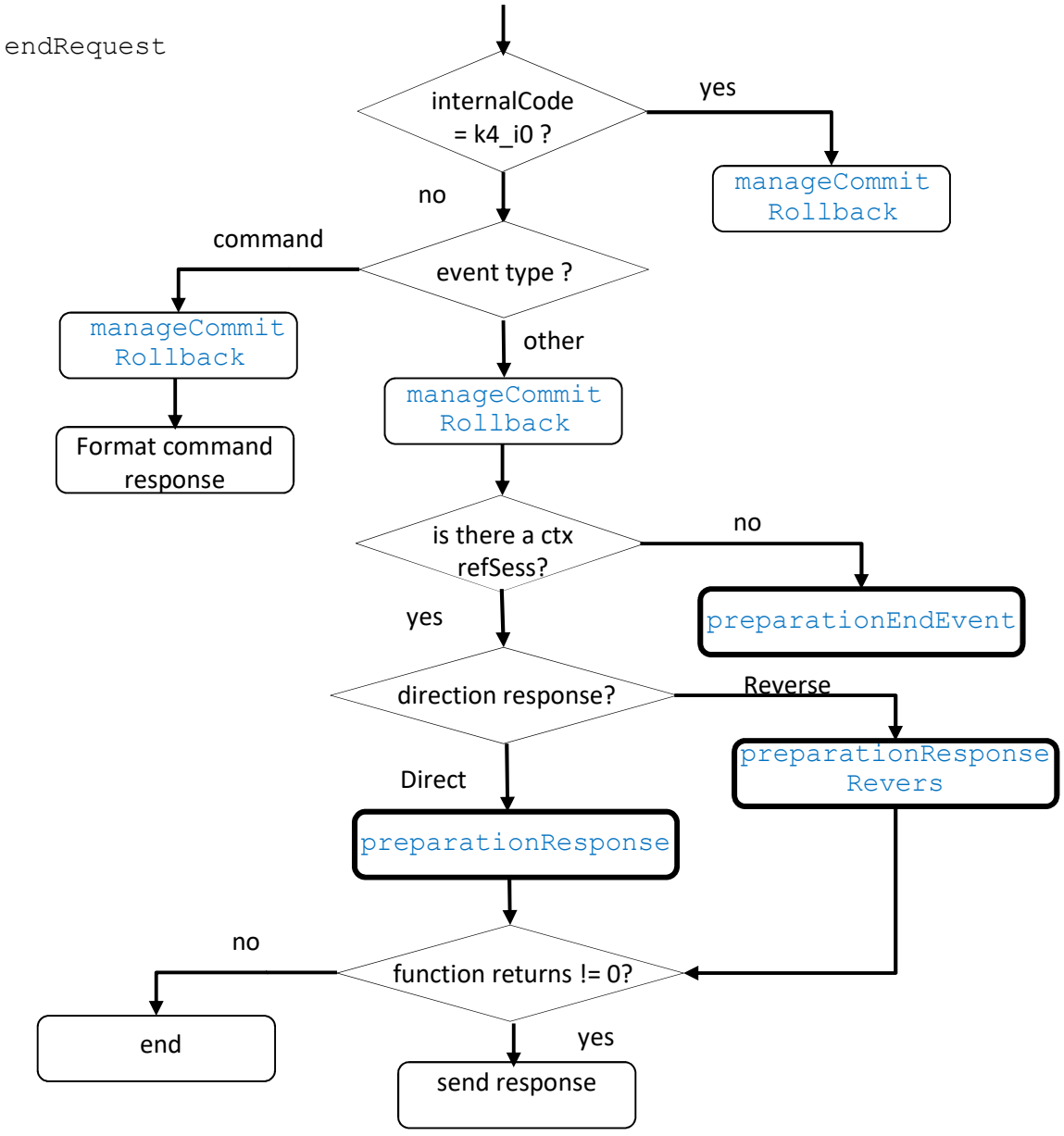

Figure 10: `endRequest` processing

PART 2: Setup when working with Tango

4. EXERCISES SETUP

When a new terminal is opened, the following steps must be performed before starting or resuming the exercises.

Step 0: check the environment variables.
- the PATH variable must contain ./bin and ./command
- the LD_LIBRARY_PATH must contain ./lib and ./module

If they are not defined properly, add the following lines in `~/.bash_profile`:

```
export PATH=$PATH:./bin:./command
export LD_LIBRARY_PATH=./lib:./module:$LD_LIBRARY_PATH
```

Step 1: go to the working directory. For the exercises, it will be `~/dev`. For the other cases, it may be the `gen` folder.

Step 2: check in the working directory that the `bin` component and the `product.lib` files are present (they may be links). If not, ask the project manager or the dev team leader to get them.

Step 3: if the working directory does not contain the tg_sys component, check the value of `TANGO_UNIC_HOME` environment variable using: `echo $TANGO_UNIC_HOME`.
If not defined, then enter this command manually (or add it in `~/.bash_profile`):

```
export TANGO_UNIC_HOME=<PATH_TO_GENDIR>
```

where `<PATH_TO_GENDIR>` is the path to the gen directory that contains all the Tango platform components.

5. HOW TO CREATE A NEW APPLICATIVE CLASS

5.1. Descriptor file

Developer needs to write one file describing its applicative component, this file called *descriptor* will allow the automatic generation of source code for a TG_CLIAPP derived class using the **`gen_prod.pl`** perl script. This script uses one special directory containing one template for a generic applicative component, it is the **pro_tmp** directory. After generating code the developer only needs to uncomment method handling events he wants to process and put the code in these methods.

In the descriptor file the developer has to fill:
- The applicative class name
- The name and type of every bus fields he wants to manipulate
- The name and type of every data he wants to store in application context
- The name and type of every data he wants to store in session context

- The name and type of every configuration fields he wants to read.

Since the generation script is a perl script, the descriptor is written using perl language and perl datatypes. Here is a commented sample of descriptor file:

```
# This is a sample descriptor for a tango applicative class

# This variable holds the destination product name
$produit = 'my_firstTangoService';

# This part describes how the text below will be replaced during generation
process from product template (pro_tmp)
# Note: will be removed
%product =
(
  'pro_tmp'  => 'my_firstTangoService',
  'protmp'   => 'myfirstTangoService',
  'PRO_TMP'  => 'MY_FIRSTTANGOSERVICE',
  'PROTMP'   => 'MYFIRSTTANGOSERVICE',
);

# ====================================================================

# Datatype mapping: entry types are defined by a letter which meaning is
provided below:

#   1 <=> 1 byte integer
#   2 <=> 2 bytes integer
#   4 <=> 4 bytes integer
#   8 <=> 8 bytes integer
#   Z <=> string
#   T <=> TG_TEMPS
#   B <=> TG_BUS
#   D <=> TG_TIMER
#   C <=> class or struct
#   X <=> array of structs
#   O <=> occurrence of bus in bus
#   P <=> occurrence of fields in bus
#   V <=> bus field containing a bus

# ====================================================================

# This section holds the definition of the configuration proxy class
# There are two possible formats for an entry, the short format and the
long format

# The short format is [ 'name', 'type', 'default value' ] or [ 'name',
'type' ]
# The long format is
# {
#  'nom'   => '<accessor name>',    # (will be renamed into 'name')
#  'champ' => '<real name>',        # (will be renamed into 'field')
#  'type'  => '<type>',
#  'def'   => '<default value>'
# }
#
# available types for config are:
#   1 <=> 1 byte integer
```

```
#    2 <=> 2 bytes integer
#    4 <=> 4 bytes integer
#    8 <=> 8 bytes integer
#    Z <=> string
#    T <=> TG_TEMPS

@cfgdef =
(
  # sample short format config accessors
  # this declares one four byte integer config accessor named bankId with 0
as default value
  [ 'bankId', '4', '0' ],
  # this declares one string config accessor named logCfgFile without
default value
  [ 'logCfgFile', 'Z', ],

  # sample long format config accessors
  # this declares one string config accessor named thisIsMyAccessor linked
to 'some technical field with spaces in it' in configuration without
default value
  { 'nom' => 'thisIsMyAccessor', 'champ' => 'some technical field with
spaces in it', 'type' => 'Z' },
);

# ====================================================================

# This section holds the definition of the bus proxy class
# There are two possible formats for an entry, the short format and the
long format

# The short format is [ 'name', 'type', 'typeout', 'default value' ] or [
'name', 'type' ]
# The long format is
# {
#    'nom'     => '<accessor name>',     # (will be renamed to 'name')
#    'champ'   => '<bus field name>',    # (will be renamed to 'field')
#    'type'    => '<type>',
#    'def'     => '<default value>',
#    'typeout' => '<output format>',
# }
#
# <accessor name>    : accessor name in c++ proxy class
# <bus field name>   : dictionary name for the field
# <type>             : type of data
# <default value>    : default value if field not present in bus
# <output format>    : output modifier, 'T' for hexascii dump, 'L' for
readonly...
#
# Available types for bus are:
#    1 <=> 1 byte integer
#    2 <=> 2 bytes integer
#    4 <=> 4 bytes integer
#    8 <=> 8 bytes integer
#    Z <=> string
#    T <=> TG_TEMPS
#    B <=> TG_BUS
#    O <=> occurrence of bus in bus
#    P <=> occurrence of fields in bus
#    V <=> bus field containing a bus
```

```
#
# There are 3 special cases for types 'O' 'P' 'V' for which the syntax are
different. See examples below:
#
@busdef =
(
  # short format samples
  [ 'oneIntegerValue', '2', '', '0'  ],
  [ 'anotherInteger',  '8'           ],
  [ 'whyNotAString',   'Z', '', '""' ],

  # long format samples
  { 'nom' => 'simpleName', 'champ' => 'complex name with spaces', 'type' =>
'B', 'def' => 'NULL' },

  # special format sample for 'O' 'P' 'V' cases
  [ 'arrayOfBus', 'O', '', [
    [ 'someField', '4' ],
    [ 'anotherField', 'Z' ],
  ]],
  [ 'justASubBus', 'V', '', [
    [ 'anythingYouWant', 'T' ],
    [ 'evenAnotherStructuredBus', 'V', '', [
      [ 'oneField', '4' ],
      [ 'anotherOne', '2' ],
    ]],
  ]],
  [ 'simpleStringArray', 'P', '', 'Z' ],
);

# ================================================================

# This section holds the definition of the session context proxy class
# There are two possible formats for an entry, the short format and the
long format

# The short format is [ 'name', 'type', 'default value' ] or [ 'name',
'type' ]
# The long format is
# {
#   'nom'      => '<accessor name>',    # (will be renamed into 'name')
#   'type'     => '<type>',
#   'init'     => '<default value>',
#   'struct'   => '<c struct name>',
#   'find'     => '<find function name>',
#   'typefind' => '<type of data used for searching>',
#   'fct'      => '<method giving searched data value>',
# }
#
# Available types for session bus are:
#   1 <=> 1 byte integer
#   2 <=> 2 bytes integer
#   4 <=> 4 bytes integer
#   8 <=> 8 bytes integer
#   Z <=> string
#   T <=> TG_TEMPS
#   B <=> TG_BUS
#   D <=> TG_TIMER
@session =
```

```
(
  [ 'oneName',   '1' ],
  [ 'something', 'Z' ],
  [ 'foo',       'D' ],
);

# ============================================================

# This section holds the definition of the request context proxy class
# There are two possible formats for an entry, the short format and the
long format

# The short format is [ 'name', 'type', 'default value' ] or [ 'name',
'type' ]
# The long format is
# {
#   'nom'      => '<accessor name>',    # (will be renamed into 'name')
#   'type'     => '<type>',
#   'init'     => '<default value>',
#   'struct'   => '<c struct name>',
#   'find'     => '<find function name>',
#   'typefind' => '<type of data used for searching>',
#   'fct'      => '<method giving searched data value>',
# }
#
# Available types for request context bus are:
#   1 <=> 1 byte integer
#   2 <=> 2 bytes integer
#   4 <=> 4 bytes integer
#   8 <=> 8 bytes integer
#   Z <=> string
#   T <=> TG_TEMPS
#   B <=> TG_BUS
#   D <=> TG_TIMER
@context =
(
  [ 'theSame',  '1' ],
  [ 'again',    '2' ],
  [ 'andAgain', 'Z' ],
);

1;
```

5.2. Generating with gen_prod.pl

To generate source code from the descriptor you have to use **`gen_prod.pl`** giving it the name of the descriptor as command line parameter.

```
gen_prod.pl lusis_tst.desc

Generating product 'lusis_tst'
Product 'lusis_tst' has been successfully generated
```

After this step, you have to move the descriptor file into the directory containing the product.

This script will create a directory tree with full product sources and headers; it generates also configuration files for the compilation system. The generated directory tree will be like this:

lusis_tst/	depend.lib		
	lusis_tst.PRJ		
	config/	logCfgFile_lusistst.xml	
	include/	lusistst_bus.h	lusistst_general.h
		lusistst_cfg.h	lusis_tst.h
		lusistst_ctx.h	lusistst_ses.h
		lusistst_def.h	pro_cliext.h.sav
	source/	lusistst_bus.cxx	lusistst_int.cxx
		lusistst_cfg.cxx	lusistst_log.cxx
		lusistst_cmd.cxx	lusistst_not.cxx
		lusistst_ctx.cxx	lusistst_rep.cxx
		lusis_tst.cxx	lusistst_req.cxx
		lusistst_dir.cxx	lusistst_ses.cxx
		lusistst_end.cxx	lusistst_tim.cxx
		lusistst_evt.cxx	pro_cliext.cxx.sav
		lusistst_ini.cxx	

The following subdirectories are created:

- config: for log messages configuration files (.xml files)
- include: for headers (.h files)
- source: for sources (.cxx files)

In the following list, filenames containing (lusistst or lusis_tst) depend on the product name in the descriptor. Unless otherwise stated these files are modifiable by the developer, those stated as non editable should be regenerated updating the descriptor.

The following files are created at the directory root:

- lusis_tst.PRJ: file for ultraedit containing the product file list
- depend.lib: config file for compilation system

The following files are created in the include directory:

- lusis_tst.h: applicative class definition, method that developer wants to overload should be uncommented in this file
- lusistst_bus.h: autobus definition **DO NOT EDIT THIS FILE**
- lusistst_cfg.h: configuration proxy class definition **DO NOT EDIT THIS FILE**
- lusistst_ctx.h: transaction context proxy class definition **DO NOT EDIT THIS FILE**
- lusistst_def.h: this header is for constants and type definitions
- lusistst_general.h: main include file, includes other file
- lusistst_ses.h: session context proxy class definition **DO NOT EDIT THIS FILE**
- pro_cliext.h.sav : used only for plugin generation, outside the scope of these exercises

The following files are created in the source directory:

- lusis_tst.cxx: constructor and other basic initialization things, not often modified
- lusistst_bus.cxx: autobus source code **DO NOT EDIT THIS FILE**
- lusistst_cfg.cxx: configuration proxy class source code **DO NOT EDIT THIS FILE**
- lusistst_cmd.cxx: This file contains methods for handling commands
- lusistst_ctx.cxx: transaction context proxy class source code **DO NOT EDIT THIS FILE**
- lusistst_dir.cxx: this file contains methods for handling direction of events (no often used)
- lusistst_end.cxx: this file contains "end of processing" methods like consignation or prepareResponse
- lusistst_evt.cxx: this file contains method to handle event (only for network adapters)
- lusistst_ini.cxx: this file contains methods for configuration, sql request, initialization and other thing like this
- lusistst_int.cxx: this file holds every function added by the developer
- lusistst_log.cxx: this file contains methods for logging (file logging)
- lusistst_not.cxx: this file contains methods for handling notifications
- lusistst_rep.cxx: this file contains methods for handling responses
- lusistst_req.cxx: this file contains methods for handling requests
- lusistst_ses.cxx: session context proxy class source code **DO NOT EDIT THIS FILE**
- lusistst_tim.cxx: this file contains methods for handling timers
- pro_cliext.cxx.sav : file used only to make plugins, outside the scope of these exercises

5.3. Updating generated class with regen_prod.sh

To modify the content of a generated proxy class you should:

- Modify the descriptor (add or remove fields or change fields)
- Use the `regen_prod.sh` command giving it descriptor's filename as command line argument

Without any other parameter the **`regen_prod.sh`** command will regenerate all proxy classes, you can choose which class you want to regenerate by adding their names on command line. For instance, the `regen_prod.sh my_descriptor.desc bus ctx` command will only regenerate the bus and transaction context proxy class. You can use the following names:

- `bus` for autobus regeneration
- `ctx` for transaction context regeneration
- `ses` for session context regeneration
- `cfg` for configuration regenereration

6. COMPILATION

Compilation in TANGO is performed by perl scripts using configuration file describing inter-module dependencies, for an applicative service the compilation system will produce one shared library containing the whole compiled class source code.

6.1. Compiling applicative class

To compile one applicative class and generate the associated shared library you have to update the following file:

- `depend.lib` : it defines dependences between applicative class and other classes (often system class, sometimes applicative libraries)

The file `depend.lib` is a perl include read by the compilation script with the perl `require` directive. To avoid errors while running compilation, depend.lib file should always end with "1;" (one and semicolon)

For an applicative component, the depend.lib file will contain at least the tg_cliapp component

Note: some dependencies may have already been defined in the product.lib file in the root directory of the generation environment; in this case there is no need of putting them in the depend.lib.

Below is an example of depend.lib file:

```
#
# Compilation dependencies configuration file

$LIBNAME  = "lusis_tst";
$ISSHARED = 1; # 1 for a shared library
$VERSION  = "0.0.1";

@DEPEND           = ("tg_cliapp");
@EXTRA_INC        = ();
$EXTRA_FLAGS      = "";
@DATABASE         = (); # ('mysql','oracle','db2')
1;
```

To compile you have to do the following:

- `cd <the_product_i_want_to_compile>`
- `compil.pl`

Available options for compil.pl are:

- `release`: compilation with optimizations and without debugging information
- `debug`: compilation without optimizations and debugging information
- `clean`: erase every .o files so everything has to be recompiled in this product

6.2. Using the compiled library

To use this library, the latter should be copied to or linked from the `module/` subdirectory of the environment, and when the module description (DECLARE_MODULE) is modified, you should regenerate the config/modules.xml file of the environment using the `bin/modules` binary.

```
bin/modules module/ > config/modules.xml
```

PART 3: Exercises

7. ACCESS TO WORKING ENVIRONMENT

7.1. Jupyter

In the Tango development training, click on Tango exercises puzzle button and then "Start My Server".

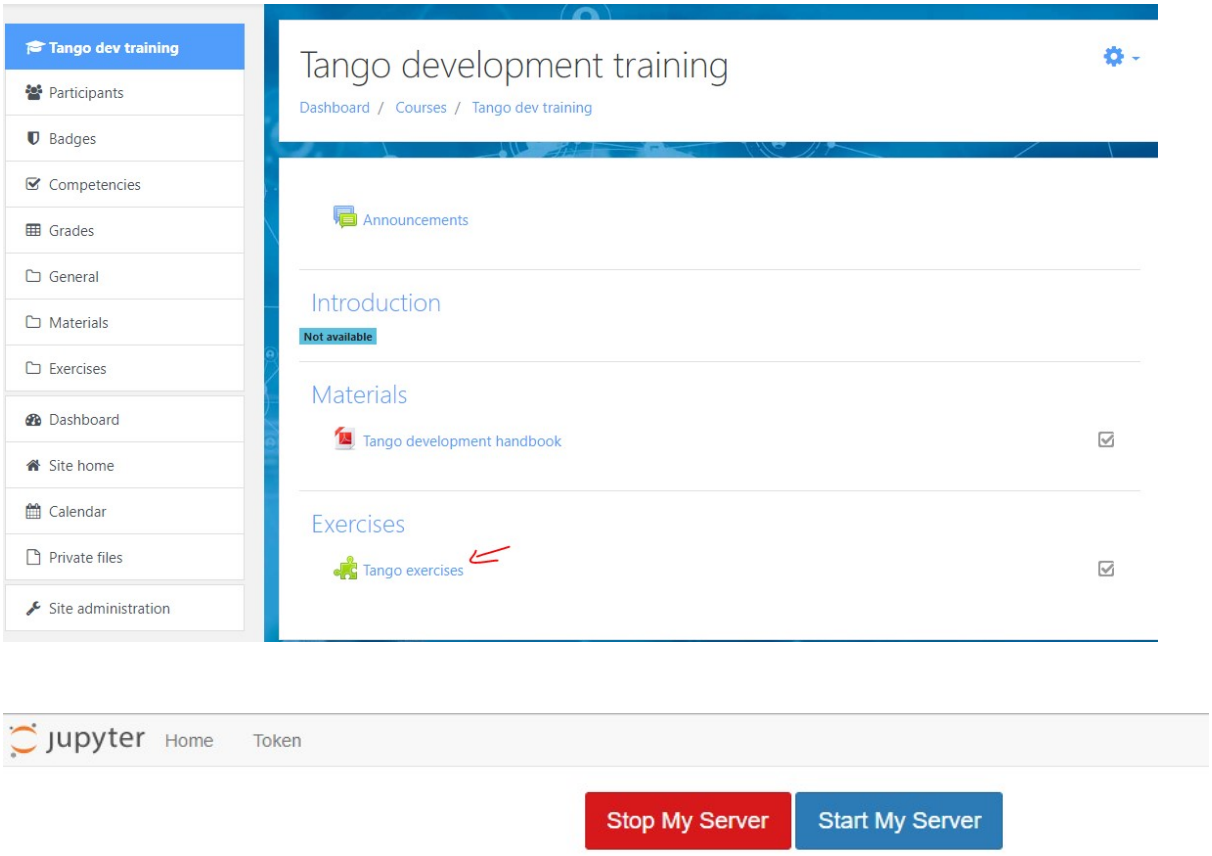

Figure 11: Access to Jupyter notebook

After a while, the Jupyter notebook opens:

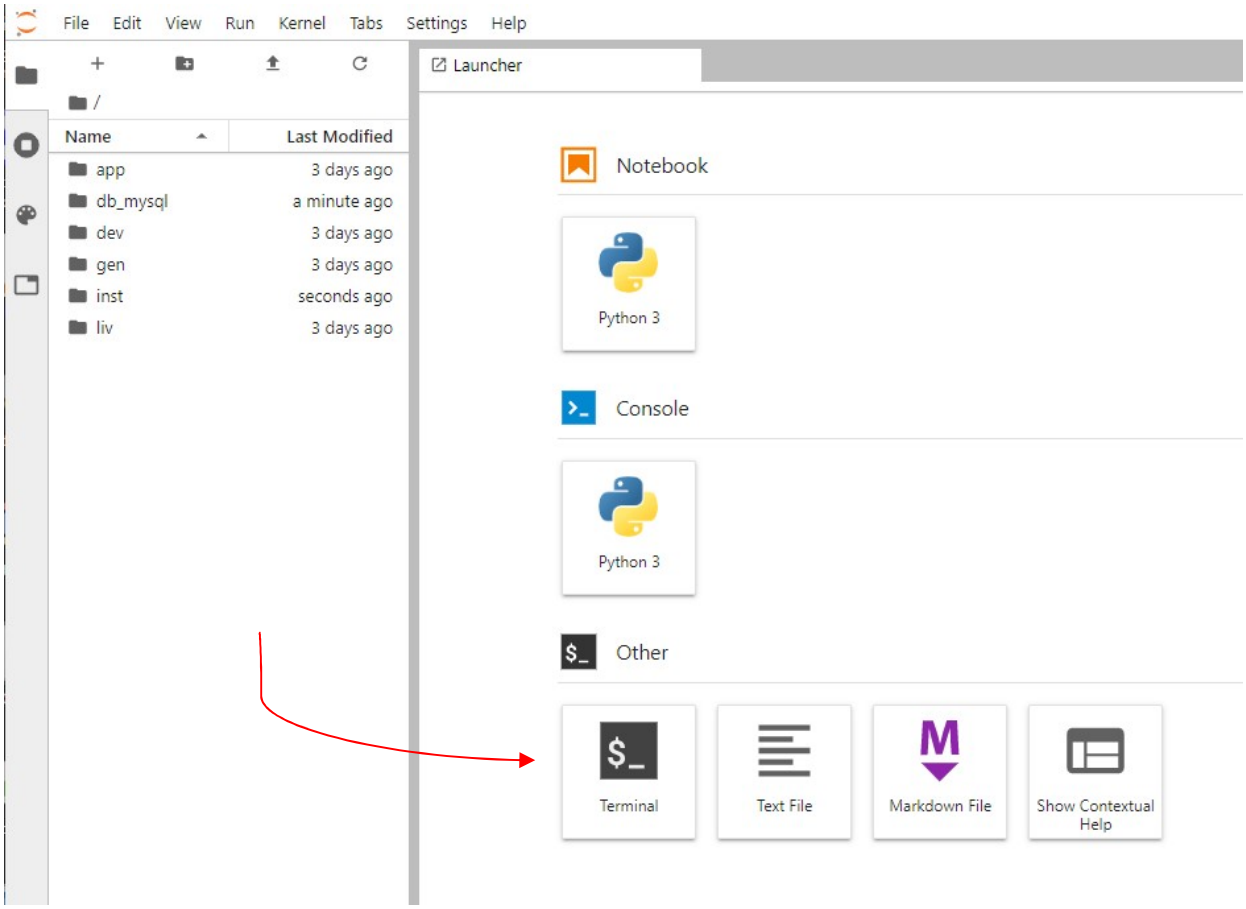

Figure 12: Access to Jupyter notebook

Click on the Terminal icon to open a new terminal session.

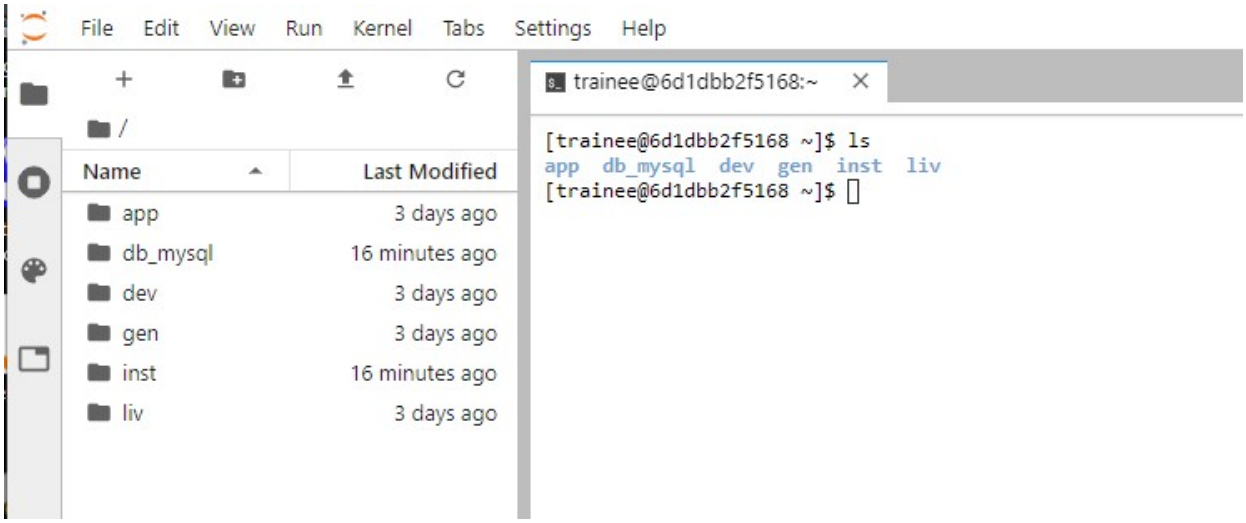

Figure 13: Terminal session content

The folders are the following:

- app: the Tango environment root directory
- db_mysql: contains the MySQL server. **DO NOT TOUCH IT**
- dev: the folder used for development. **ALL EXERCISES MUST BE DONE IN THIS FOLDER**
- gen: the generation directory. Contains all the libraries and header files for the Tango application. **DO NOT TOUCH IT**, but it is possible to view the API methods from the header files
- inst: the installation log directory. **DO NOT TOUCH IT**
- liv: the delivery directory. **DO NOT TOUCH IT**

7.2. Git with Gitea

The URL to access the git repository is: https://elearning.lusis.net:3000/

Figure 14: Gitea login page

Click on the "Sign in" button and **enter your credentials provided by your tutor**.

Then you should arrive to the home page. Click on the "+" button and then New Repository to create a new folder

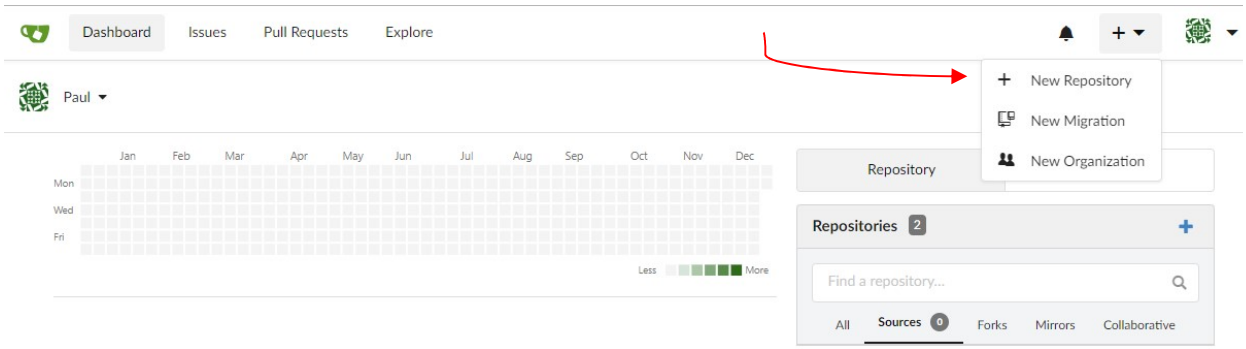

Figure 15: Gitea home page

Create a new repository called lusis_tst, the name of the module you will create for the exercises and tick the "Make Repository Private" cell before creating the repository.

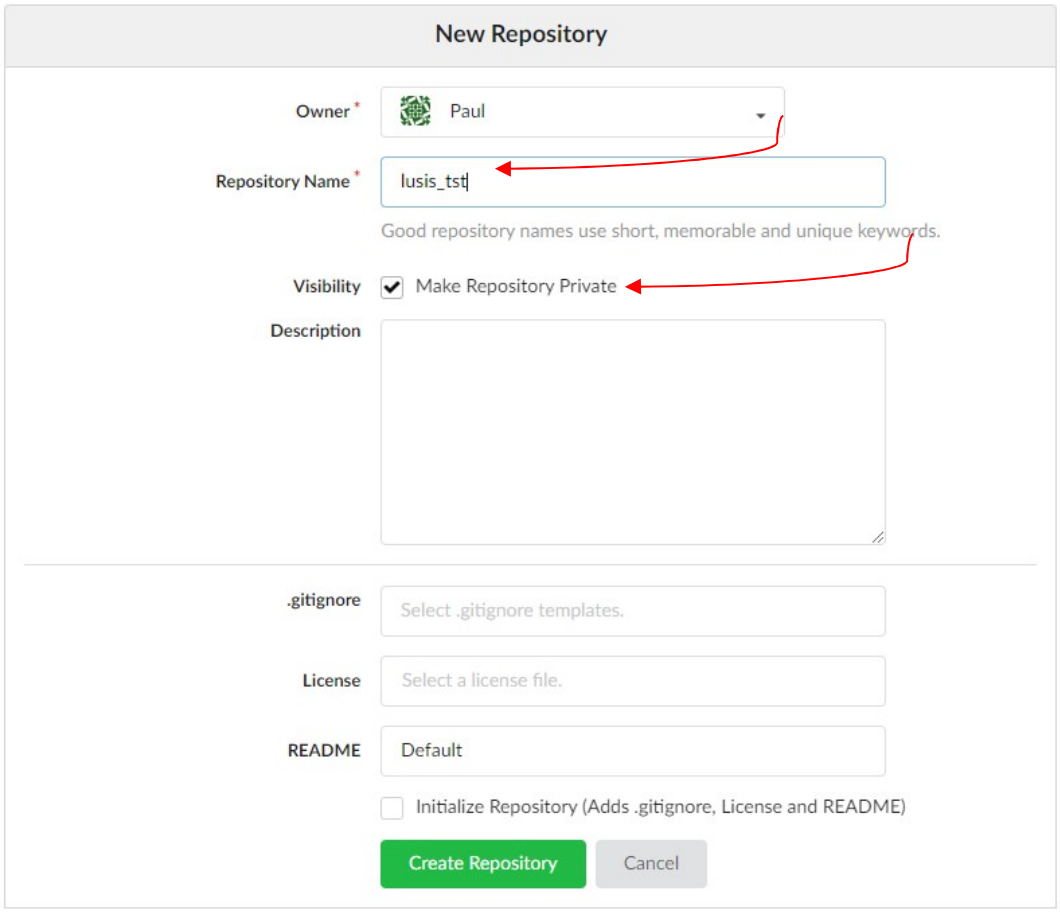

Figure 16: Gitea create new repository page

Once the repository is created, right access must be given to the tutor. Click on the "Settings" button, go to "Collaborators" tab and add your tutor with Read access

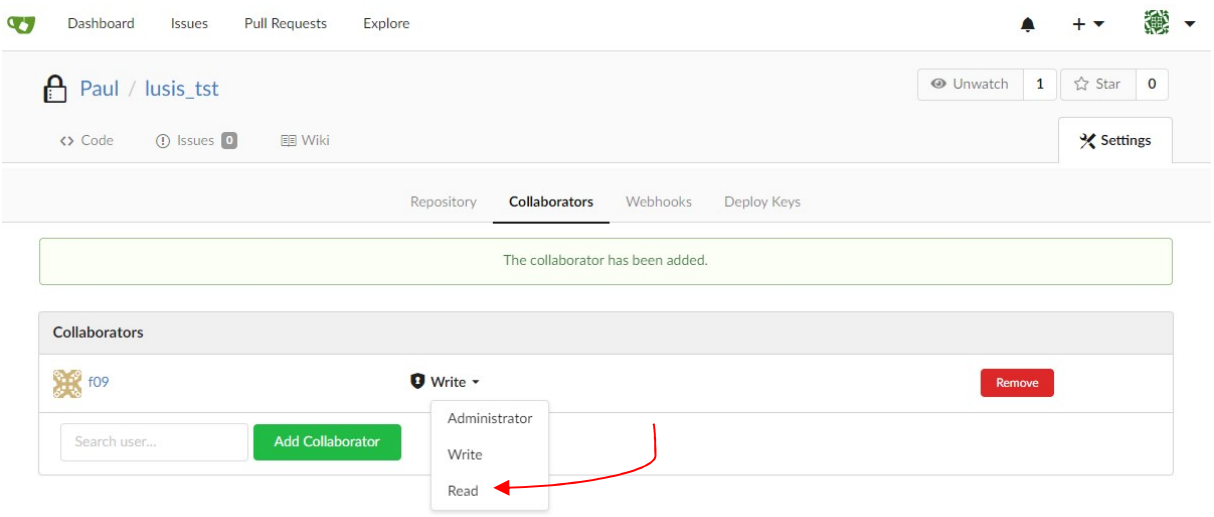

Figure 17: Gitea add collaborators page

8. REQUIREMENTS FOR ALL EXERCISES

The application developer will create an **inherited class** from TG_CLIAPP class and overload some TG_CLIAPP methods according to their needs.

There are 12 exercises in total. As each one is based on the previous one (apart from exercises 11 and 12), it is mandatory to do the exercises in order. For each exercise, you will be asked to do the following, in addition to each exercise specific requirements:

- Understand the exercise new concept
- Understand the transaction flow if applicable
- Implement the solution in compliance with the Tango Dev Handbook [3].
- Take care of error cases
- Write unit tests (except for exercises 11 and 12)
- Test your module in the runtime environment and keep track of your test cases
- Commit and Tag under git, the versioning control system used by Lusis

8.1. Writing unit tests

Refer to document **Unit tests for Tango development** for writing unit tests with Tango, available here: https://elearning.lusis.net/mod/page/view.php?id=37

8.2. Using git

If you have never used git, try the `man git` command or do some research on the Internet.
You need to know how to perform the following actions:

- Import new directories and files to git
- Checkout the master or branch version of the module from git
- Checkin or commit to git
- Put a tag

- Check the status, history and log of changes for each file
- Managing a branch including: create a branch tag, move to the branch tag and move back to the master branch

9. EXERCISE 1: Answer to one request

9.1. Objective

In this exercise, you will implement a component that answers an authorization request. At first, this component will always provide the same response.

The component should check if following fields are present and if their value is numerical:

- *MTI* (it must be equal to `4000`)
- *cardNumber*
- *transactionAmount*
- *terminalId*

If one field is missing or contains invalid content, the component will reply with the additional *resultCode* field to the bus set to value `2900`. If everything is OK, the component will add the *resultCode* field in the bus with value `0` and the *authorizationNumber* field with value `"123456"`.

9.2. Steps

- Draw the Tango application system
- Study the transaction flow
- Identify required bus fields
- Write the service descriptor with **lusis_tst** as the name of Tango component.
- Generate code using **`gen_prod.pl`**
- Write unit tests
- Overload `algoRequest() and preparationResponse()`
- Compile the library
- Add the library to the runtime environment
- Update of the module definition file (modules.xml)
- Start the environment and check if it starts without problem by watching in the log (file log)
- Manually write an XML test bus like this:

```
<bus>
  <field name="MTI" type="I">4000</field>
  <!-- to be completed... -->
</bus>
```

- Send the test bus to the service using the **`sendBus.sh`** script in the environment

- Test every possible case

10. HOW TO USE DATABASE

TG_CLIAPP allows the use of databases; it provides a way to prepare SQL queries during the initialization of the applicative class. In the `startQuery()` method, the developer initialize the `mm_queryText` map associating an unique number with SQL query text. After calling this method, TG_CLIAPP will prepare all the queries according to the RDBMS engine and store proxy class for SQL request (DB_QUERY) in the `mm_pQuery` map using the same key as provided with SQL text.

The queries are implemented like this:

In `lusistst_def.h` file:

```
// Query map constants
enum
{
  k4_reqSelectCard,
  k4_reqOppo
};
```

In `lusistst_ini.cxx` file:

```
/**
 * sql request declaration
 */
void LUSIS_TST::startQuery()
{
  const byte1*      ka_funcName = "LUSIS_TST::startQuery";
  trace(tg_traceProTrc,mc_rcvMskTraMsg, ka_funcName, (void*)0, (byte4)0);

  TG_CLIAPP::startQuery();

  mm_queryText[k4_reqSelectCard] = "SELECT state FROM CARD WHERE cardNumber
= ?";
  mm_queryText[k4_reqOppo]  = "SELECT reason FROM OPPO WHERE cardNumber =
?";

}
```

Usage:

```
  DB_QUERY *lcp_query;

  lz_cardNumber =  mcp_bus->cardNumber();

  lcp_query = mm_pQuery[k4_reqSelectCard];
  lcp_query->bindCol(1, l4_cardState);
  lcp_query->bindParam(1, lz_cardNumber);
  lcp_query->execute();
```

```
if (!lcp_query->fetch())
{
  mcp_bus->resultCode(k4_rc_unknownCard);
  m4_codeInterne = k4_x1;
  return;
}
```

11. EXERCISE 2: Access database (1/2)

11.1. Objective

In this exercise, you will modify the previous program to add some controls by reading data from tables. You have to check the card state and the account balance.

You will add the *resultCode* field with these respective values in these cases:

- `2901` if card is unknown (not present in table CARD) or inactive (state != 1)
- `2902` if card is present in table CARD and if CARD.accountBalance is lower than the *transactionAmount* of the request
- `0` if card is present in table CARD and if CARD.accountBalance is greater or equal to *transactionAmount* of the request

You will add the *accountBalance* field to the bus in these cases:

- *resultCode* `2901`: no *accountBalance* field
- *resultCode* `2902`: *accountBalance* in answer holds the database value
- *resultCode* `0`: *accountBalance* in the response equals the database balance value minus *transactionAmount*. For example, if you have 500€ on your account (provided by the field CARD.ACCOUNTBALANCE) and you wish to withdraw 100€, you will answer 400 in *accountBalance*. This new value of *accountBalance* will also be recorded in the CARD table.

The *authorizationNumber* field should be present in the answer only if *resultCode* equals `0`.

11.2. Database connection information

Host: 127.0.0.1
Database type: MySQL
Database name: TDTANGO
Database user: tango
Database password: tango

11.3. Steps

- Study the transaction flow
- Write unit tests
- Create the SQL queries
- Identify required bus fields
- Update service descriptor

- Regenerate code using **`regen_prod.sh`**
- Update `algoRequest`() and `preparationResponse`()
- Compile the library
- Update of the service configuration to add database parameters
- Start the environment and check if it starts without problem by watching in the log (file log)
- Test every possible case, including non-regression tests

12. HOW TO SEND SUB-REQUESTS AND PROCESS RESPONSES

Sometimes, a component has to send a sub-request to another applicative component before responding properly to the initial request. In order to do that, the request context is used. TG_CLIAPP will automatically retrieve the initial request context when receiving a sub-request response.

The developer must:

- define each data that has to be saved
- save these data in the request context
- retrieve them when receiving the response associated to the sub-request

These data must be defined in the descriptor and they are not related to the Tango dictionary.

When processing a request that involves a sub-request emission, the developer must prevent the sending of the automated response. In order to do that, the global variable `m4_codeInterne` must be set to `k4_i0` value (this constant is defined at TANGO system level) when executing `algoRequest()` method or `algoRequestRevers()` method. Then he/she can send a sub-request by executing `envoiRequest()`.

When receiving the sub-request response, TG_CLIAPP will call `algoReponse()` method or `algoResponseRevers()` method, depending on the response direction which is defined by the method `whichDirectionResponse()`. When all the processing is done, the response of the main request will be issued back to the initial sender. The current autobus is used to create this response, so it may be necessary to modify it to set the response. The `preparationResponse()` method is called before sending the response in order to allow the modification the response bus one last time.

13. EXERCISE 3: Send a sub-request

13.1. Objective

In this exercise, you will modify the previous component in order to send a sub-request to get the authorizationNumber field value. The sub-request bus must contain the following fields:

- *MTI*=`6000`
- *cardNumber* (must be equal to the initial request)
- *transactionAmount* (must be equal to the initial request)

- *terminalId* (must be equal to the initial request)
- *accountBalance* (the one calculated in exercise 2)

The sub-request will be sent to a stub which will always respond the same thing: *authorizationNumber* = "`987654`"

After receiving the sub-request response, *MTI* will be reset to `4000`.

13.2. Steps

- Study the transaction flow
- Write unit tests
- Update `algoRequest()` and `preparationResponse()` methods.
- Implement `algoResponse()` method.
- Compile the library
- Start the environment and check if it starts without problem by watching in the log (file log)
- Test every possible case, including non-regression tests

13.3. Extension: handle dispatcher error

In the environment root directory, launch the following command to stop the ISSUER process:

```
$stopproc.sh ISSUER NOSURV
```

- Run your test cases and observe the difference of behavior. Analyze the transaction flow.
- Override `algoResponseErreurDisp()` to add the dispatcher error code in the bus *resultCode*

14. EXERCISE 4: Access database (2/2) - Record transaction

14.1. SQL commit / rollback

Regardless of the query type (SELECT, INSERT, UPDATE, DELETE), the SQL queries must be committed to or rollbacked from the database at the end of the event processing by using either:

```
    mcp_db->commit() ;
    mcp_db->rollback();
```

Note: commit and rollback should be done one per transaction and not on every SQL query.

14.2. Objective

In this exercise, you will modify the previous program to record the following transaction information:

- *cardNumber*
- *transactionAmount*
- *authorizationNumber*
- *accountBalance*
- *terminalId*
- *resultCode*
- *MTI*

If an error occurs during recording, a response with a field *resultCode* = `3099` will be expected.

Recording will be done in the TRANSACTIONS table. Any missing field in the bus during the recording should be set to the default value `-1`.

The timestamp column will be filled with the current time of the recording. To have this exact time, an instance of `twist::DateTime` or `twist::DateTimeTz` will be used. The API of these two classes are defined in the `twist` component under `twist/include/tw_datetime.h` and `twist/include/tw_datetimeTz.h`

Note: in older versions of tg_sys V6, the `TG_TEMPS` class will be used instead and initialized with the `.now()` method. This class is defined in `tg_sys/include/tg_temps.h`

Transaction recording should be done as late as possible before replying to the sender. Therefore, the `consignation()` method is a good place to perform this operation.

14.3. Steps

- Study the transaction flow
- Modify source code
- Compile the library
- Restart environment
- Tests

14.4. Extension: commit and rollback management

- Alter the issuer stub file so that it does not send the *authorizationNumber* anymore.
- On response, if the *resultCode* returned by the issuer is OK but there is no *authorizationNumber*, then throw an exception with the code `1234`.
- Override `manageCommitRollback()` to rollback when an exception is thrown
- Test the different cases by modifying the issuer stub file.

15. HOW TO HANDLE FILE LOGGING

TG_CLIAPP provides a standard way to write in Tango log file. Messages issued to the log server are defined in an XML configuration file called the **logCfgFile**.

Before sending the record to the log server, TANGO's platform seeks a matching entry inside memory tables built from the **logCfgFile**.

Each record in the **logCfgFile** contains the following tags:

Tag	Presence	Multiplicity	Description
<LOG>	mandatory	1	Root tag
<INCLUDE>	optional	0..N	Include other logCfgFile. This is used for derivate components that requires their parent log message
<TYPE>	mandatory	0..N	Contains the log message information
<TYPE>.<CLASS>	optional	0..1	Applicative class name
<TYPE>.<CODE>	mandatory	1	Key used to find the record from implementation code when calling `logCliApp(p4_code)`
<TYPE>.<EVT>	mandatory	1	Applicative log code of the log record
<TYPE>.<SEVERITY>	mandatory	1	Severity of the log record. It is ranged from 0 to 4: 0: critical 1: major 2: error 3: warning 4: informative
<TYPE>.<LABEL>	mandatory	1	Label (text description) of the log record
<TYPE>.<ORIGIN>	mandatory	1	Reserved for future use
<TYPE>.<USERNUMBER>	mandatory	1	Reserved for future use
<TYPE>.<USERTEXT>	mandatory	1	Reserved for future use
<TYPE>.<VERBOSITY>	optional	0..1	If absent, the log record will always be present in the log file. If a value is provided, the record will be logged only if this value is lower than the verbosity level filter set in the environment configuration.
<TYPE>.<DATAFORMAT>	optional	0..1	void: print each MLLP arguments values separated by '/'. %E: print each MLLP argument_name=value separated by '/'. %L: print each MLLP arguments values separated by '/'. %nX / %vX (X being a number): print the value/name of an argument. See below for more information.
<TYPE>.<DATADESC>	optional	0..1	Each argument name separated by '/'. This field is needed when using %E or %vX of DATAFORMAT.
<TYPE>.<COMMENT>	optional	0..1	Free comment text

Table 1: logCfgFile description

In the implementation code, the `logCliApp(p4_code)` or `tmpAppExcept(p4_code)` method can be used by the developer to emit one message to the log file.

User data part is filled by the developer either:

- In the `mz_logData` property of TG_CLIAPP before calling `logCliApp` or `tmpAppExcept`.
- By calling the alternative prototypes of `logCliApp(p4_code, :(arg1, arg2, …))` and `tmpAppExcept(p4_code, MLLP(arg1, arg2, …))`, where arg1, arg2 are the data to be displayed, typically a bus field name and value. MLLP stands for Multiple Log List Parameters.

16. EXERCISE 5: Log in TANGO file

16.1. Objective

In this exercise you will modify the previous program to had signalizations in the log when *cardNumber* is unknown or when *accountBalance* is exceeded:

Case	Card unknown	Balance exceeded
CODE	2000	2001
SEVERITY	2	3
LABEL	Unknown card	Balance exceeded
ORIGIN	LUSIS_TST	LUSIS_TST
DATA	*cardNumber*	*cardNumber/accountBalance/transactionAmount*

16.2. Steps

- Create and fill in the logCfgFile_lusistst.xml
- Update the code to add signalizations
- Write unit tests
- Compile the library
- Update the environment configuration
- Start environment and check if everything is ok
- Test every possible case, including non-regression tests and check the log messages.

16.3. Extension: use MLLP arguments

Do the same exercise using MLLP arguments instead of `mz_logData`.

Add the following information in the logCfgFile_lusistst.xml:

Case	Card unknown	Balance exceeded
DATAFORMAT	%L	%L
DATADESC	*cardNumber*	*cardNumber/accountBalance/transactionAmount*

17. HOW TO HANDLE OPERATIONAL COMMANDS

17.1. Command definition

Command definition is called in the config() method:

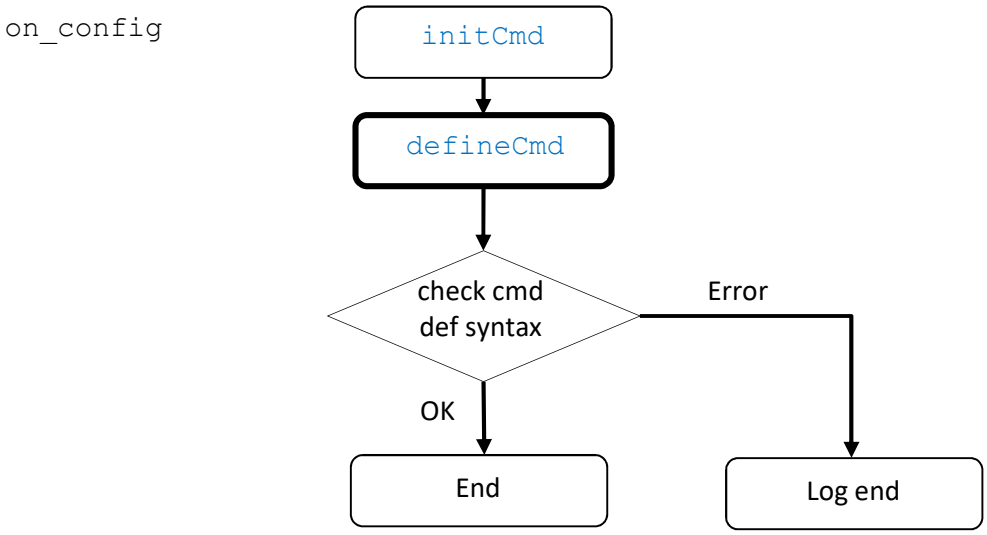

Figure 18: `on_config` processing for commands

The command definition syntax in Tango is similar to Linux commands

```
/// @brief Defines the command
void PRO_TMP::defineCmd()
{
  const char*      ka_funcName = "PRO_TMP::defineCmd";
  trace(tg_traceProTrc, tg_traceMskNone, ka_funcName, (void*)0, (byte4)0);

  // defines the service name and brief description in
  // service command helper
  mz_serviceName.assign("PRO_TMP");
  mz_serviceBrief.assign("pro_tmp"); // brief description of service

  // commands sample to be readapted

  string lz_cmdDef;
```

```
  lz_cmdDef = ""
"Command name:\n"
"  cmdTest1 - Command test 1\n"
"\n"
"Usage:\n"
"  cmdTest1\n"
"\n"
"Description:\n"
"  Simple command, no option, no param\n";
  REGISTERCMD(lz_cmdDef, &PRO_TMP::procCmdTest1); // PRO_TMP::procCmdTest1
is an internal method that is executed when command cmdTest1 is called

  lz_cmdDef = ""
"Command name:\n"
"  cmdTest2 - Command test 2\n"
"\n"
"Usage:\n"
"  cmdTest2 (-a|-t) [-d=<date>] <P1> [<P2>]\n"
"\n"
"Options:\n"
"  -a            Option a\n"
"  -t            Option t\n"
"  -d            Date\n"
"\n"
"Parameters:\n"
"  <date>        [YYYYMMDD]\n"
"  <P1>          Parameter 1 [an5]\n"
"  <P2>          Parameter 2 [n1] [default: 1]\n"
"\n"
"Description:\n"
"  Complex command with options and parameters\n"
"\n"
"Examples:\n"
"  Add some usage examples here (optional)\n"
"\n"
"Notes:\n"
"  Add some notes here (optional)\n";
  REGISTERCMD(lz_cmdDef, &PRO_TMP::procCmdTest2);
}
```

17.2. Command processing

`on_command`

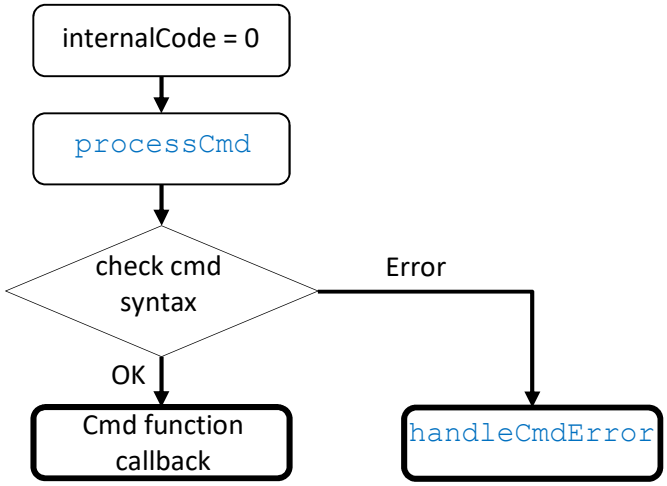

Figure 19: `on_command` processing

After syntax analysis, the `mm_tokenCmd` map is updated to indicate whether or not each keyword is present, value of each of its parameters as character string and the sum of its decimal parameters.

The TG_CLIAPP member data `mz_nonParser` contains unrecognized keywords.

Examples with previous map initialization:

```
// The code samples below are examples of implementation of cmdTest1 and
cmdTest2

// byte4 PRO_TMP::procCmdTest1(string &pzr_cmdResp)
// {
//   const char* ka_funcName = "PRO_TMP::procCmdTest1";
//   trace(tg_traceProTrc, tg_traceMskNone, ka_funcName, ( void* )0, (
byte4 )0);
//
//   pzr_cmdResp.assign("Command 1 processed"); // define response here
//
//   return k4_a1; // return internal code
// }
//
// byte4 PRO_TMP::procCmdTest2(string &pzr_cmdResp)
// {
//   const char* ka_funcName = "PRO_TMP::procCmdTest2";
//   trace(tg_traceProTrc, tg_traceMskNone, ka_funcName, ( void* )0, (
byte4 )0);
//
//   pzr_cmdResp.assign("Command 2 processed with:\n");
//
//   // this method loops over all options. To check if a specific option
is set, check for the boolean in the mm_option map:
//   // if (mm_options.find("-d") != mm_options.end()) {...}
```

```
//   pzr_cmdResp.append("  options: ");
//   map<string,bool>::const_iterator lc_option = mm_options.begin();
//   for (; lc_option != mm_options.end(); ++lc_option)
//     if (lc_option->second)
//       pzr_cmdResp.append(lc_option->first).append(1,'/');
//   pzr_cmdResp.append("\n");
//
//   // this method loops over all parameters and fixed words. To retrieve
the values of a specific parameter, use syntax:
//   // vector<string> &lvr_values = mm_params.at("<P1>");
//   pzr_cmdResp.append("  params: ");
//   pzr_cmdResp.append("\n");
//   map<string, vector<string> >::const_iterator lc_param =
mm_params.begin();
//   for (; lc_param != mm_params.end(); ++lc_param)
//   {
//     pzr_cmdResp.append("    ").append(lc_param->first).append(": ");
//     for (size_t i = 0; i < lc_param->second.size(); i++)
//       pzr_cmdResp.append((lc_param->second)[i]).append(1,'/');
//     pzr_cmdResp.append("\n");
//   }
//
//   return k4_a1;
// }
```

17.3. Command syntax error handling

The processing of the command is done in the `algoCommand()` method. You can prepare the answer to the command in `algoCommand()`. Answer is formatted with one keyword system like for syntax analysis.

```
/// @brief Handles command error
/// @param[in,out] pzr_cmdResp Command response string
/// @return internal code
byte4 PRO_TMP::handleCmdError(string &pzr_cmdResp)
{
  const char* ka_funcName = "PRO_TMP::handleCmdError";
  trace(tg_traceProTrc, tg_traceMskNone, ka_funcName, ( void* )0, ( byte4
)0);

  // pzr_cmdResp contains the command error message
  // here, write additional error processing
  // optionally, reformat pzr_cmdResp
  // and return internal code
  return k4_a1;
}
```

18. EXERCISE 6: Operational commands in TANGO

18.1. Objective

In this exercise, you will modify the previous program to add counters: the total number of transactions and the accumulated amount of these transactions. You will also add an operational command allowing following options:

- To print the total number of transactions: `-total`
- To print the accumulated amount: `-accum`
- To reset every counter: `-reset`
- It should be possible to combine options like for example: `-total -accum -reset`

When resetting counters, you have to add one line in the log with the following information:

Case	Counter reset
CODE	2002
SEVERITY	4
LABEL	"Counter reset"
ORIGIN	"LUSIS_TST"
DATA	total / accum

18.2. Steps

- Update the program to add counters
- Update the program to add commands
- Write unit tests
- Compile the library
- Start environment and check if everything is ok
- Test your program, including non-regression tests and check the log messages.

19. HOW TO HANDLE TIMERS

19.1. Definition

Delays are handled through timers:

- Setting a timer means to specify the delay before it will trigger, you can do it using `TG_TIMER::set(p8_delay)` and the delay is expressed in milliseconds.
- Resetting a timer means to deactivate it, you can do it using `TG_TIMER::reset()`.

You cannot set a timer to repeat. You have to set it again every time it expires.

There are four kinds of timers:

- User timers: these timers are an extension of object timers. Like object timers, they are properties of the applicative class, but they also contain a void* pointer that allows the developer to store any user data tied to the timer.
- Object timers: these timers are properties of the applicative class; they can be used to achieve asynchronous tasks like doing something every X amount of time.
- Request timers: these timers are members of the request context. They are declared by using a 'D' type in your product descriptor; they allow you to handle non-response timeout for example, in this case you have to think about resetting the timer upon response reception.
- Session timers: like request timers they are tied to the session context and shall be declared by adding a 'D' type member in your application descriptor; they allow you to handle inactivity timers for example.

19.2. Examples

Object timer examples:

```
TG_TIMER mc_purgeTimer;
TG_TIMER mc_scanTimer;
```

Context timer examples:

```
@context =
(
  [ 'responseTimer'                , 'D'                          ],
  [ 'requestTimer'                 , 'D'                          ],
);
```

19.3. Timer properties

19.3.1. Timer identification

Each timer has a unique number, when it expires, only its number is known. To determine which timer has expired (and its type), the developer shall code the 4 following functions:

- `isTimerUser()`
- `isTimerObject()`
- `isTimerCtx()`
- `isTimerSes()`

These functions return:

- 0 if there is no such timer found
- The timer's number if found (this number is defined by the developer).

```
byte4 LUSIS_TST::isTimerObject(byte8 pc_whichTimer)
{
  if (pc_whichTimer == byte8((size_t)& mc_purgeTimer))
    return k4_purgeTimerId;
```

```
  if (pc_whichTimer == byte8((size_t)& mc_scanTimer))
    return k4_scanTimerId;

  return 0;
}

byte4 LUSIS_TST::isTimerCtx(byte8 pc_whichTimer)
{
  mcp_ctx->findrequestTimer(pc_whichTimer);
  if (mcp_ctx->exist())
     return k4_requestTimerId;

  mcp_ctx->findresponseTimer(pc_whichTimer);
  if (mcp_ctx->exist())
     return k4_responseTimerId;

  return 0;
}
```

The value returned by the `isTimerXXX()` methods is passed as an argument to the `algoTimerXXX()` methods.

```
void LUSIS_TST::algoTimerObjet(byte4 p4_timer)
{
  switch (p4_timer)
  {
    case k4_purgeTimerId:
      // Handle mc_purgeTimer
      break;
    case k4_scanTimerId:
      // Handle mc_scanTimer
      break;
   }
}
```

19.3.2. Request timer additional features

The direction concept applies also on request timers, the `whichDirectionTimer()` method is called to get the direction of the timer and then it will call either `algoTimerCtx()` or `algoTimerCtxRevers()` methods to process the timer.

After processing a response timer, if the internal code is not set to `k4_i0`, `preparationResponse()` is called and an answer is issued to the initial sender.

19.3.3. User and object timers additional features

User and object timers need to be tied to the TG_CLIAPP object instance when they are instanciated.

To achieve this, use the `sefRefCli()` method:

```
void LUSIS_TST::start()
{
  mc_purgeTimer.setRefCli(getRefCli()); // binds the purge timer to the
LUSIS_TST instance
```

```
  mc_scanTimer.setRefCli(getRefCli());
}
```

19.4. Timer processing

on_timer

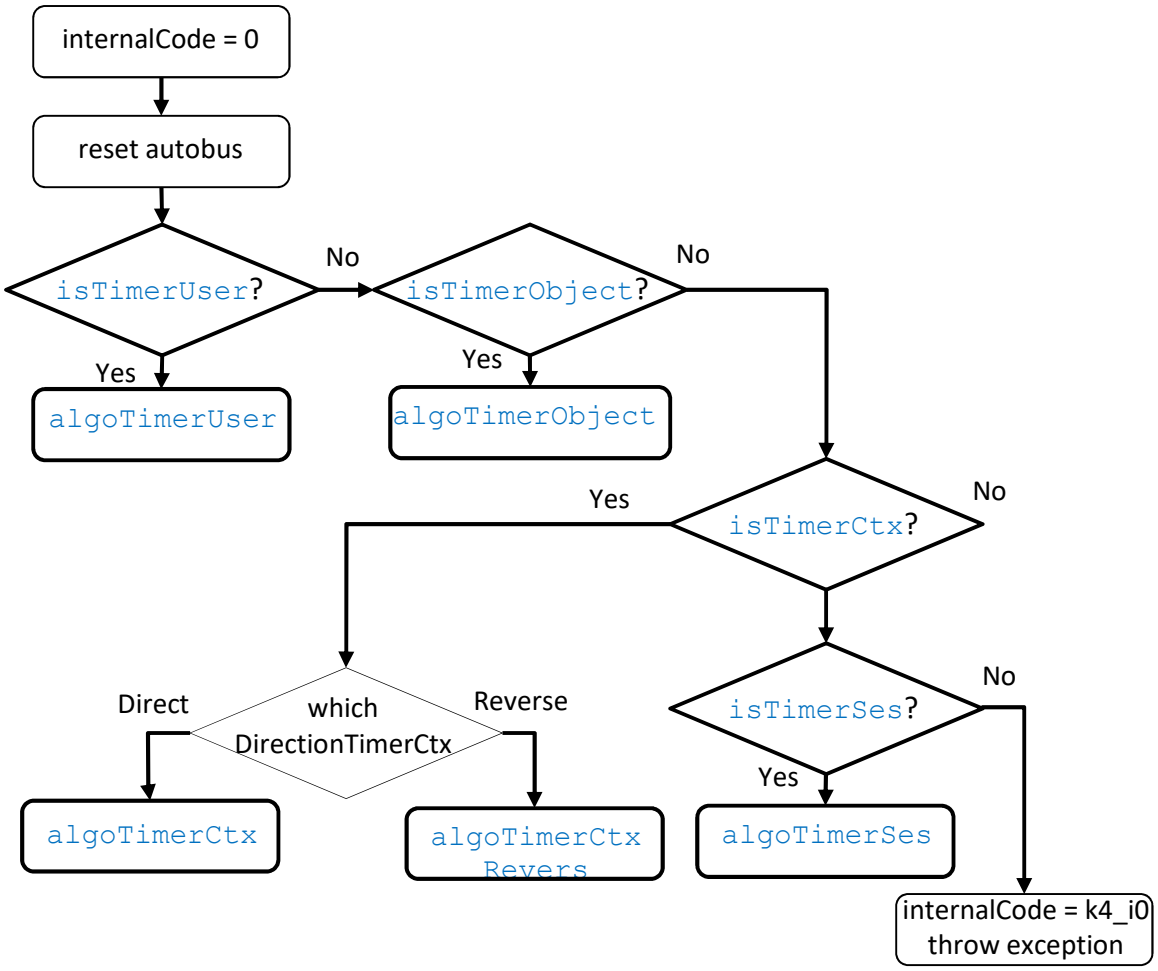

Figure 20: on_timer processing

20. EXERCISE 7: Context timer

20.1. Objective

In this exercise, you will modify the previous program to add a request timer when sending the sub-request defined in exercise 3.

If the timer expires before response reception, a response will be sent to the acquirer with a *resultCode* equal to `3096`.

20.2. Steps

- Study the transaction flow
- Write unit tests
- Update service descriptor
- Regenerate applicative class with **`regen_prod.sh`**
- Implement the `algoTimerCtx()` and `isTimerCtx()` methods
- Compile the program
- Restart environment
- Test

21. EXERCISE 8: Object timer

21.1. Objective

In this exercise, you will modify the previous program to add an object timer which will be set when the application starts, this timer will be used to send a notification with the total number of transaction and the accumulated amount of these transactions at regular intervals.

The notification should be sent every 15 seconds (15000 milliseconds) and should contain the following fields:

- *MTI* = `4100`
- *transactionAmount* = accumulated amount of transactions during this interval
- *transactionCounter* = total number of transactions during this interval

As this notification message is not a transaction, it should not be recorded in the TRANSACTIONS table.

21.2. Steps

- Study the transaction flow
- Write unit tests
- Update service descriptor
- Regenerate applicative class with **`regen_prod.sh`**
- Implement the `algoTimerObject()` and `isTimerObject()` methods
- Compile the program
- Restart environment
- Test

22. EXERCISE 9: Access TANGO configuration

22.1. Objective

In this exercise you will modify the previous program to read the notification delay from the configuration file.
To achieve this you will use the following field:

- configuration field name: *inactivityDelay*
- Type: `byte4`
- Default value: `15000`

22.2. Steps

- Update service descriptor
- Regenerate applicative class with **`regen_prod.sh`**
- Compile the program
- Update configuration. Use `vfield` to avoid modifying the dictionary.
- Restart environment
- Test

23. EXERCISE 10: Manage session context

23.1. Objective

In this exercise, you will modify your applicative component to:

- Add the processing for *MTI* = `3900` (ATM connection request). Only one field is present in the request bus: *terminalId*. If the ATM is already connected or did not disconnect properly, you have to respond with *resultCode* = `2903`.
- Add the processing for *MTI* = `3901` (ATM disconnection request). Only one field is present in the request bus: *terminalId*. If ATM is not connected, you have to respond with *resultCode* = `2904`.
- Create a session context during the ATM connection request, save the total number of transactions as well as the total transaction amount sum for the current ATM connection.
- Update session context during each authorization request (*MTI* = `4000`).
- Modify exercise 5 command in order to add an optional keyword `-terminal` that specifies which ATM's counter to return or reset. If `-terminal` keyword is missing, the command returns the global counter or resets all counters according to other options.Steps
- Study the transaction flow
- Write unit tests
- Update service descriptor
- Regenerate applicative class with **`regen_prod.sh`**

- Update `algoRequest()` method
- Compile the program
- Restart environment
- Test

24. TG_BATCH

24.1. Presentation

TG_BATCH class is a C++ class that hides the TANGO system classes for standalone programs.

Batch programs are often used in the following situations:

- Read data from tables and generate a report in a file or by sending message(s) to TANGO
- Read an external file and populate data in tables

Batch programs bear the following properties:

- They are launched through a command line with possibly some parameters.
- They can connect to a database
- They can send requests or notifications synchronously via the InterTango component (or ITG for short). However, they are not connected to the dispatcher.
- They can log and trace data
- They can exit with a specific code ; they can therefore be integrated in a .sh or cron script
- They can handle timers
- They can reuse existing TANGO libraries

24.2. How to create a new batch

24.2.1. Structure

A batch component has the following organization:

`my_batch/`	`depend.lib`	
	`my_batch.PRJ`	
	`config/`	`logCfgFile_mybatch.xml`
	`include/`	`my_batch.h`
		`mybatch_bus.h`
		`mybatch_general.h`
	`source/`	`mybatch_bus.cxx`
		`mybatch_int.cxx`
	`exec.lib`	
	`EXEC/`	`my_batch.cxx`

The quickest way to achieve this is to use **`gen_prod.pl`** on a descriptor file and then, delete unused ones.

24.2.2. Setup

A sample of tg_batch derivate is available here: https://elearning.lusis.net:3000/Lusis/test_batch

24.2.3. Environment configuration

A batch program needs two configuration files: `MY_BATCH.ini` and `MY_BATCH.xml` files

Content of `MY_BATCH.ini` file:

```
# PA-DSS activation flag set 1 to enable, 0 to disable (default disabled)
padss = 0
```

Content of `MY_BATCH.xml` file:

```
<bus>
   <include file='config/include/common.xml'/>

   <field type='IH' name='TASK_CFG trace level'>0</field>
   <field type='A'
name='logCfgFile'>config/log/logCfgFile_atmsimu.xml</field>
   <field type='A' name='netAddress'>localhost</field>
   <field type='A' name='netAddressExt'>PORTNUMBER</field>
   <field type='I' name='TNR'>30000</field>
</bus>
```

Note that tg_batch can't use config/local/etc/services. The port number has to be defined as a number.

25. EXERCISE 11: ATM simulator

25.1. Objective

In this exercise, you will create a new batch called `atm_simu` that will send the request bus to your component instead of using `sendBus.sh`.

`atm_simu` takes as parameters the field *cardNumber* and *transactionAmount*. Since `atm_simu` represents an ATM, its *terminalId* will have an arbitray fixed value of `99`.

Launching the program through this command in the root environment directory

```
./bin/atm_simu -c <cardNumber> -a <transationAmount>
```

Will send the following data request:

- *MTI* equal to `4000`
- *cardNumber*
- *transactionAmount*
- *terminalId*

If the command is invalid, a helper must be displayed with the correct syntax.

25.2. Steps

- Study the transaction flow
- Write the program descriptor
- Generate code using **`gen_prod.pl`**
- Implement the component
- Compile the program
- Add the binary to the runtime environment under bin/ directory
- Test

Note: your test should fail at this stage because you are trying to perform a withdrawal while your ATM is not connected from exercise 10. You can use `sendBus.sh` to connect ATM 99 for now and implement the ATM connection and disconnection request later in the batch.

PART 4: Appendix

26. Mass compilation

This paragraph extends the notion of compilation of paragraph 6 COMPILATION.

There is a way to compile every product in one generation environment using only one command. This procedure is based on a file called **versions** which should be at the root of the generation environment along with **product.lib** file. This file contains the list of all products and for each the compilation mode is specified.

Here is an example of versions file:

```
# CVS - Do not modify this header ! #
# $Date: 2016/02/09 10:58:48 $ #
# $RCSfile: versions,v $ #
# $Revision: 1.2 $ #
# #(@) $Name:  $ #
#LSC - Project Tango (ptf) - Product Tango training - Version V01_00 -
2016-02-09T11:58:00+01:00
#Format:
# [ "component name (checkout directory)", "CVS revision/tag", "action",
"type", "cvsroot", "CVS module name or path" ],
# action = C : clean / R : release / D : debug
# type = l : lib / m : module / e : executable
# cvsroot = if omitted use environment variable
@versions = (

  # Tango components
  [ 'bin',               'V02_12', '-', '-'],
  [ 'pro_tmp',           'V02_02', '-', '-'],
  [ 'texp',              'V02_37', '-', '-'],
  [ 'tg_installer',      'V01_33', '-', '-'],
  [ 'tg_sys',            'V06_21', 'R', 'l'],
  [ 'tg_cliapp',         'V03_09', 'R', 'm'],
  [ 'tg_unittests',      'V01_11', '-', '-'],
  [ 'tg_daemon',         'V01_02', 'R', 'm'],
  [ 'tg_batch',          'V02_14', 'R', 'm'],
  [ 'tg_db_mysql',       'V02_28', 'R', 'm'],
  [ 'tg_expreval',       'V01_05', 'R', 'm'],
  [ 'tg_disp',           'V06_10', 'R', 'm'],
  [ 'tg_hyp',            'V04_35', 'R', 'm'],
  [ 'tg_simu',           'V02_19', 'R', 'm'],
  [ 'tg_utils_gen',      'V03_09', 'R', 'e'],
  [ 'tg_logger',         'V01_05', 'R', 'e'],
  [ 'tg_logFileWriter',  'V01_01', 'R', 'm'],
  [ 'tg_logTangoWriter', 'V01_02', 'R', 'm'],
  [ 'tg_traceSrv',       'V01_02', 'R', 'e'],
  [ 'tg_tcpsvm',         'V02_55', 'R', 'm'],
  [ 'tg_sessvm',         'V01_74', 'R', 'm'],
  [ 'tg_itgsvm',         'V01_14', 'R', 'm'],

  # Other components
  [ 'lusis_dic',         'V01_00', '-', '-'],

  # Tango components
```

```
  [ 'lusis_tst',          'V01_00', 'R', 'm'],

  # Other components
  [ 'cfg_tdtango',        'V01_00', '-', '-'],
  [ 'cmd_tdtango',        'V01_00', '-', '-'],

  # Database components
  [ 'mpd_tdtango',        'V01_00', '-', '-'],
);
```

The previous sample used git tags for versioning and 'R' stands for 'Release' so it will issue 'compil.pl clean' and 'compil.pl release' for every module listed in this file.

To launch the mass compilation you have to use the **`comp.pl`** script.

27. Optional extension of exercise 2: tg_table

This extension is reserved for internal collaborators of Lusis.

You will now replace your code to use tg_table to access the database. In order to use this, manipulated table should have a byte8 unique ID.

Refer to the following link for more information: http://lsc.lusis/tango-docs/?p=134

- Describe your table (see "table descriptor")
- Generate classes for the described table (`gen_db.pl`).
- Include the generated tg_line class in your module.
- Declare your class as member (see in TG_CLIAPP)
- Update constructors (see in TG_CLIAPP)
- Initialize built-in database request (see in TG_CLIAPP)
- Use tg_line::get(id) to load a line.
- Replace your code to use your tg_line class.