Fesa Equipment Links

1 Introduction

In FESA environment most classes are developed as standalone classes. But sometimes it is necessary to define some relationships between Fesa Classes. Those relationships are called Equipment-links.
Currently we can define three different types of Equipment Links:

- Friend relationship
- RDA relationship
- Interface relationship

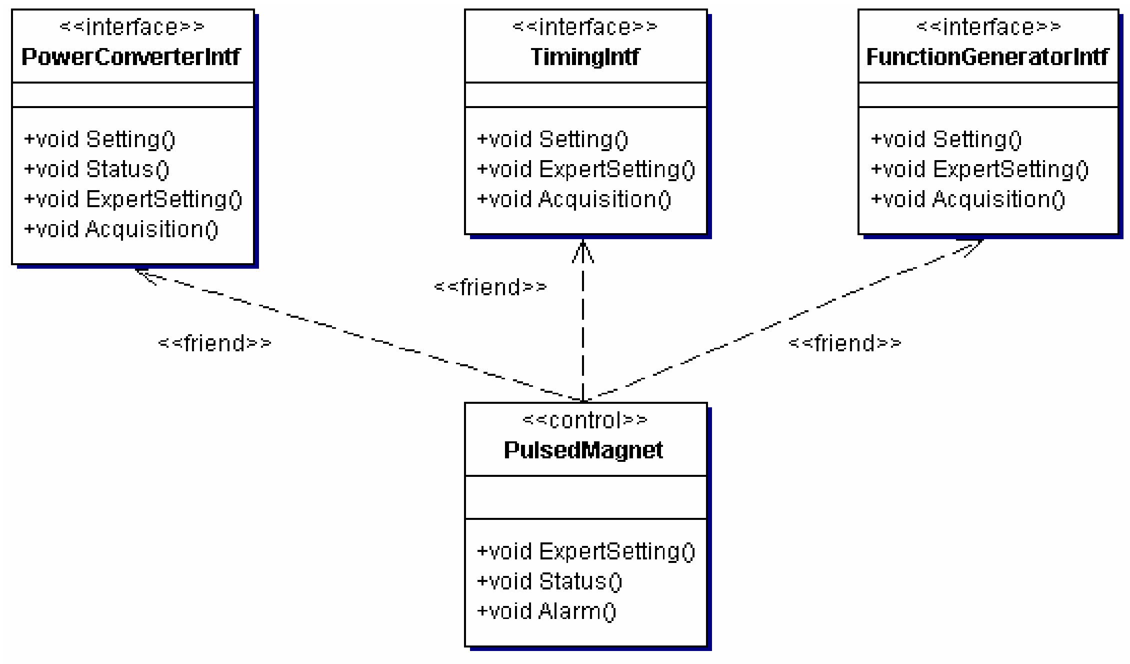
2 Friend equipment link

The “Friend” keyword indicates that using this kind of relationship a Fesa class can give direct access to its private Data Model to the Friend Fesa class. In fact it’s the same concept as the one defined by the C++ language.

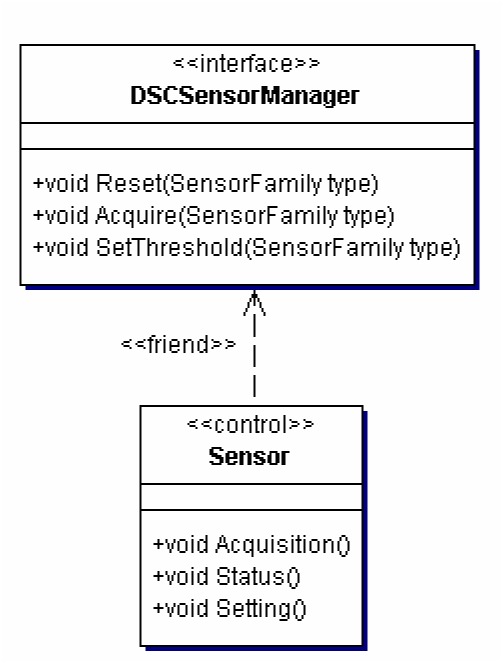
2.1Use cases

Friend Equipment-links allow different assemblies of equipment-classes, but let us restrict to the use-case that we want to promote.
We have identified two typical use-cases where this kind of relationship can be used:

- **Multiple Interfaces**: sometimes, equipments are so complex that they will generate very complex interfaces if we try to implement them into a single Fesa Class. In this case it comes very natural to break this complexity into well defined sub-functionalities which can be easily implemented into different Fesa classes. By doing this you open the possibility to have clients looking to this complex equipment through different specialized views.



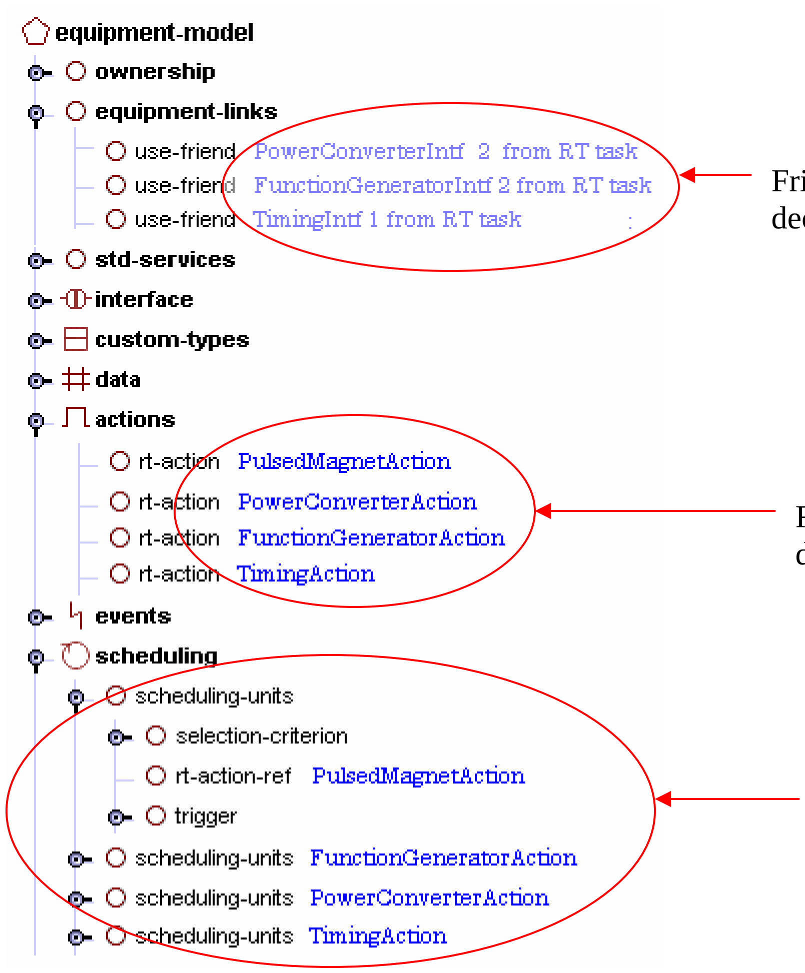
- **Mediator**: some Fesa classes can have a lot-of devices instances (100, 200,..) deployed per FEC. In this case we can imagine client like for example “supervisor system” which want to talk with a “manager” deployed as a single instance on each FEC, instead of talking directly with all instances, to execute actions common to all the devices or at least common to a family of devices



2.2 Design

The core control class (Sensor or PulsedMagnet in the above figures) usually orchestrates real-time activity on behalf of the whole set of tightly-coupled classes: the central control accesses the hardware, processes raw-data and posts processed-data into the surrounding class's devices.
Therefore the core control class design should contains the complete description in terms of interface, data, actions, events, scheduling and equipment-links where you have to specified all the friends links.
In other hand all the surrounding classes should have a restraint design which contains only the interface data and actions containing only server-actions.
The following pictures shows clearly typical design for the core Control class and for one of the surrounding class in this case FunctionGeneratorIntf

Control class design:



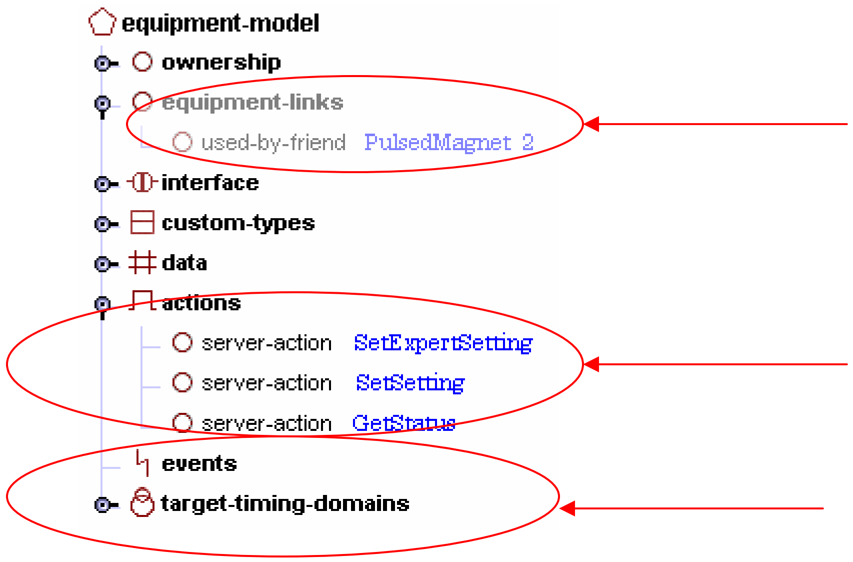
```
equipment-model
  ownership
  equipment-links
    use-friend  PowerConverterIntf  2  from RT task
    use-friend  FunctionGeneratorIntf 2 from RT task
    use-friend  TimingIntf 1 from RT task
  std-services
  interface
  custom-types
  data
  actions
    rt-action  PulsedMagnetAction
    rt-action  PowerConverterAction
    rt-action  FunctionGeneratorAction
    rt-action  TimingAction
  events
  scheduling
    scheduling-units
      selection-criterion
      rt-action-ref  PulsedMagnetAction
      trigger
    scheduling-units  FunctionGeneratorAction
    scheduling-units  PowerConverterAction
    scheduling-units  TimingAction
```

Friend Interfaces declaration

RealTime Action definition

Scheduling definition

FunctionGeneratorIntf class design



```
equipment-model
  ownership
  equipment-links
    used-by-friend  PulsedMagnet  2
  interface
  custom-types
  data
  actions
    server-action  SetExpertSetting
    server-action  SetSetting
    server-action  GetStatus
  events
  target-timing-domains
```

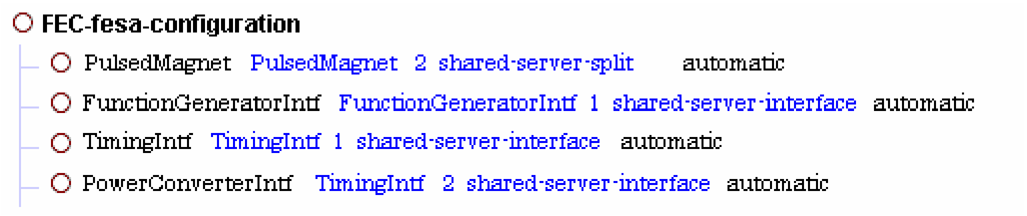
Publish the fact that this Fesa class is not a stand-alone class but is an interface of PulsedMagnet Fesa class

Only ServerAction

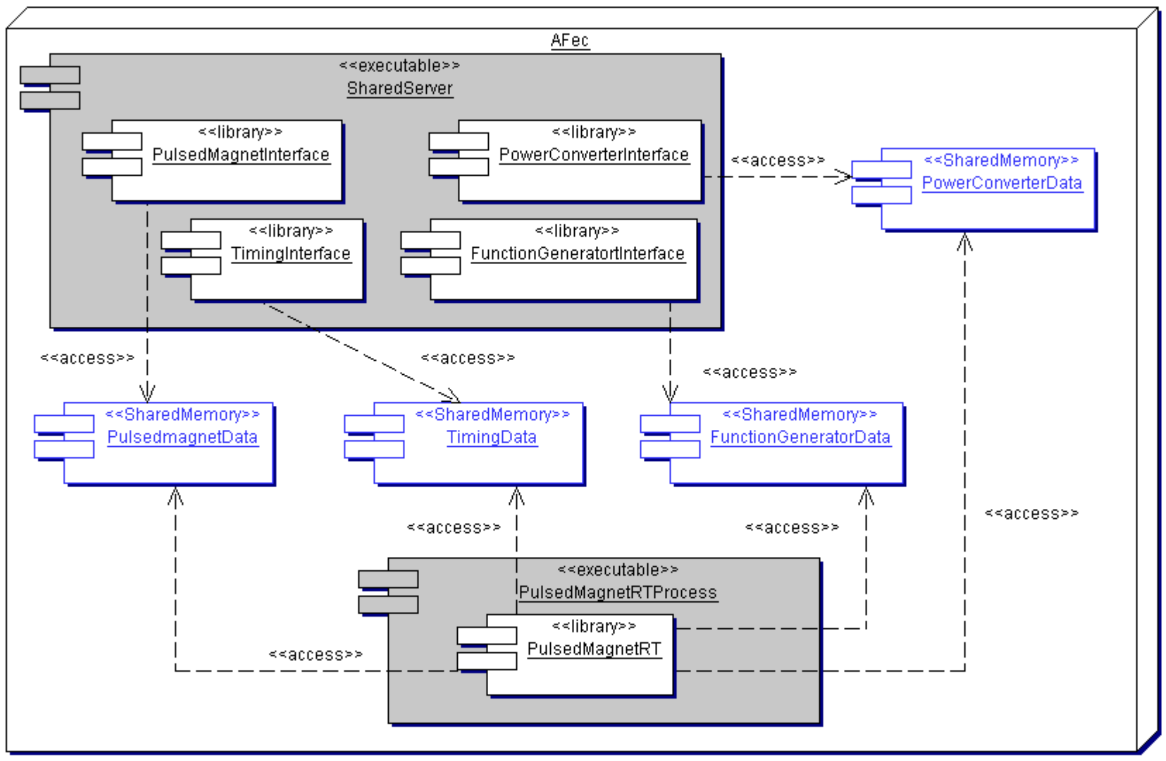
No events (empty node) and no scheduling node

2.3Deployment

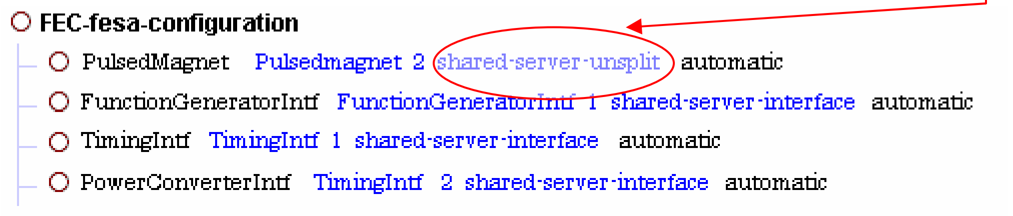
For this kind of Friend Link we recommend strongly the following deployment



- FEC-fesa-configuration
  - PulsedMagnet PulsedMagnet 2 shared-server-split automatic
  - FunctionGeneratorIntf FunctionGeneratorIntf 1 shared-server-interface automatic
  - TimingIntf TimingIntf 1 shared-server-interface automatic
  - PowerConverterIntf TimingIntf 2 shared-server-interface automatic



Since the thread priority are not correctly managed in the release 2.8, we have to deploy the control class in mode split to be able to set the priority level of the RT part of the control class (in our example PulsedMagnet). But as soon as this point will be fixed, everything could be deployed in a single process and for this, the deploy mode of the PulsedMagnet should be



- FEC-fesa-configuration
  - PulsedMagnet Pulsedmagnet 2 shared-server-unsplit automatic
  - FunctionGeneratorIntf FunctionGeneratorIntf 1 shared-server-interface automatic
  - TimingIntf TimingIntf 1 shared-server-interface automatic
  - PowerConverterIntf TimingIntf 2 shared-server-interface automatic

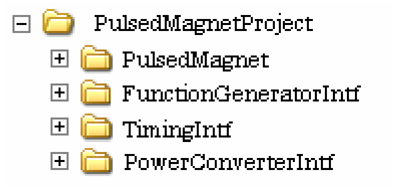
2.4 Development

2.4.1 Directories hierarchy

Create a local directory structure into which you will extract all four classes from the CVS repository:

```
cd PulsedMagnetProject
Fesa Setup PulsedMagnet 0<scratch/edit>
Fesa Setup FunctionGeneratorIntf 3 <scratch/edit>
Fesa Setup TimingIntf 1 <scratch/edit>
Fesa Setup PowerConverterIntf 0 <scratch/edit>
```

After those steps, you should have such directory hierarchy:

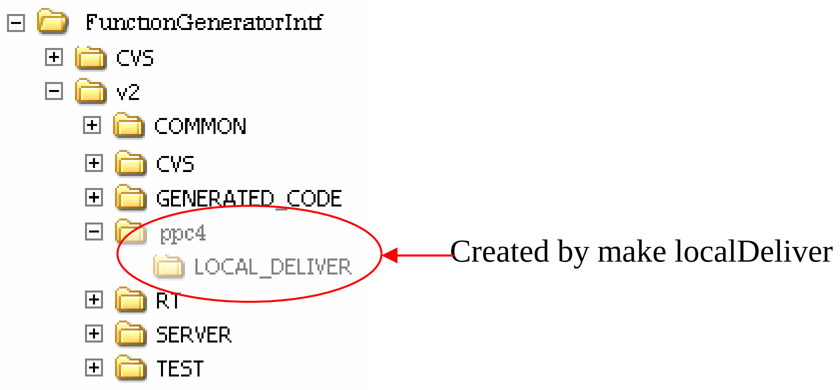


```
PulsedMagnetProject
    PulsedMagnet
    FunctionGeneratorIntf
    TimingIntf
    PowerConverterIntf
```

Then development activities can start on each class and you can proceed as usual.
The only difference is that, at a certain time in the development phase, you will need to see the DataModel of each interface from the control class which is in this case PulsedMagnet. For this point, you have to do in each surrounding classes:

```
cd FunctionGeneratorIntf/v3
make CPU=ppc4 localDeliver
cd PowerConverterIntf/v3
make CPU=ppc4 localDeliver
cd TimingIntf/v3
make CPU=ppc4 localDeliver
```

Nb: as usual CPU=ppc4 is not necessary because it’s the default platform if nothing is specified
Nb: this step must be executed only once and not each time you evolve your class because LOCA_DELIVER contains a series of links and not a copy of the files.
After this step you will see that in each surrounding classes a new directory [CPU]/LOCAL_DELIVER which contains all the symbolic links required by the Friend link.
FunctionGeneratorIntf directory hierarchy is shown below:

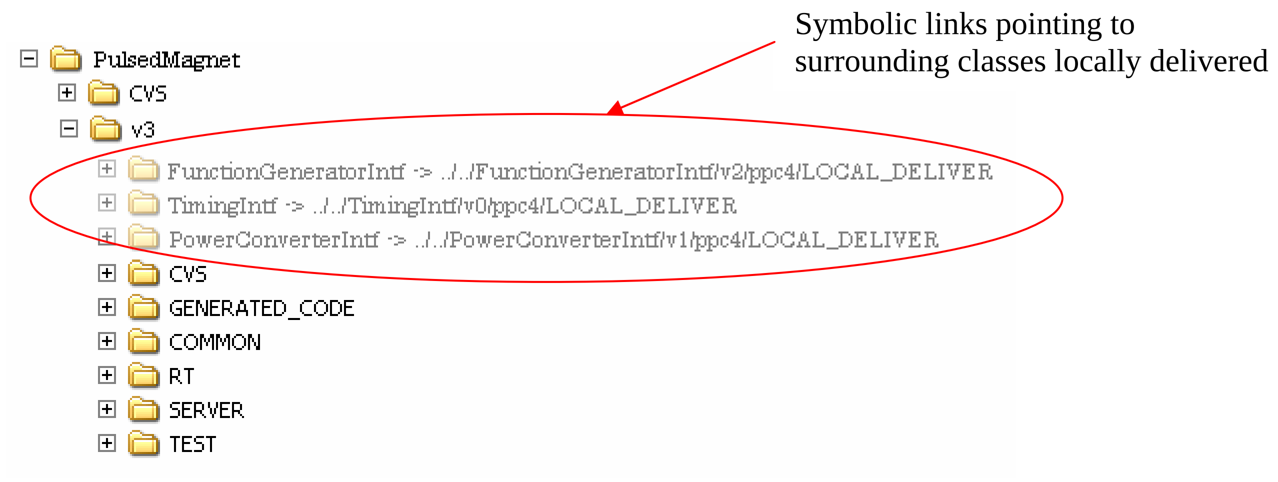


```
FunctionGeneratorIntf
    CVS
    v2
        COMMON
        CVS
        GENERATED_CODE
        ppc4
            LOCAL_DELIVER
        RT
        SERVER
        TEST
```

From now surrounding classes are ready to be visible from the control class. Then you have to establish some links to be able to see them:

```
cd PulsedMagnet/v3
ln –s ../../FunctionGeneratorIntf/v2/ppc4/LOCAL_DELIVER FunctionGenerator
ln –s ../../PowerConverterIntf/v1/ppc4/LOCAL_DELIVER PowerConverterIntf
ln –s ../../TimingIntf/v0/ppc4/LOCAL_DELIVER TimingIntf
```

The result is the following in the PulsedMagnet directory

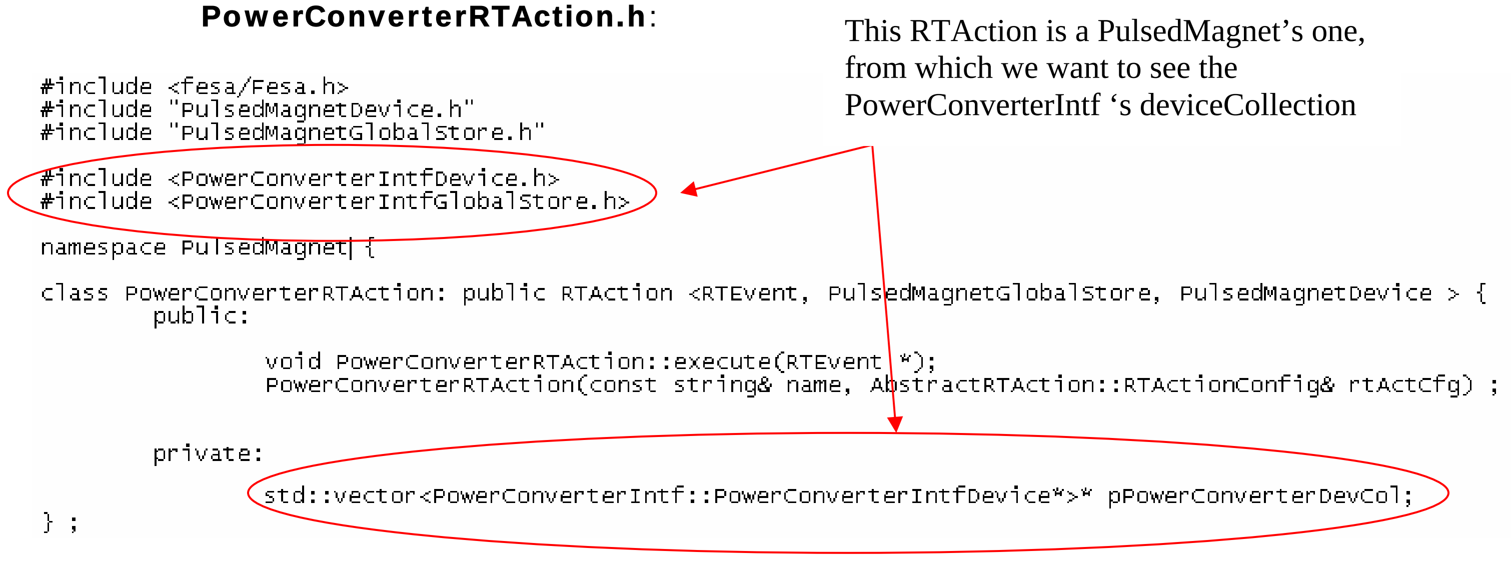

Symbolic links pointing to surrounding classes locally delivered

```
PulsedMagnet
    CVS
    v3
        FunctionGeneratorIntf -> ../../FunctionGeneratorIntf/v2/ppc4/LOCAL_DELIVER
        TimingIntf -> ../../TimingIntf/v0/ppc4/LOCAL_DELIVER
        PowerConverterIntf -> ../../PowerConverterIntf/v1/ppc4/LOCAL_DELIVER
        CVS
        GENERATED_CODE
        COMMON
        RT
        SERVER
        TEST
```

2.4.2 Source development

We have seen that the control class has defined some dedicated RTActions to process and store the data for the surrounding classes. So this means that from RTActions of the control class we need to access the different deviceCollection of the surrounding classes. The below picture show some code extraction of the PowerConverterRTAction which is in charge to process and store the data into the deviceCollection of the PowerConverterIntf Fesa class

PowerConverterRTAction.h:


This RTAction is a PulsedMagnet's one, from which we want to see the PowerConverterIntf 's deviceCollection

```
#include <fesa/Fesa.h>
#include "PulsedMagnetDevice.h"
#include "PulsedMagnetGlobalStore.h"

#include <PowerConverterIntfDevice.h>
#include <PowerConverterIntfGlobalStore.h>

namespace PulsedMagnet {

class PowerConverterRTAction: public RTAction <RTEvent, PulsedMagnetGlobalStore, PulsedMagnetDevice > {
        public:

                void PowerConverterRTAction::execute(RTEvent *);
                PowerConverterRTAction(const string& name, AbstractRTAction::RTActionConfig& rtActCfg) ;


        private:

                std::vector<PowerConverterIntf::PowerConverterIntfDevice*>* pPowerConverterDevCol;
} ;
```

PowerConverterRTAction.cpp:

```
PowerConverterRTAction::PowerConverterRTAction(const string& name, AbstractRTAction::RTActionConfig& rtActCfg) :
   RTAction<RTEvent, PowerConverterIntfGlobalStore, PowerConverterIntfDevice>(name, rtActCfg){

   //initialized once the private reference on the PowerConverterIntf deviceCollection
   pPowerConverterDevCol = PowerConverterIntf::PowerConverterIntfDevice::getDeviceCollection();
}

void PowerConverterRTAction::execute(RTEvent * pEv){

   // iterate on the PulseMagnet deviceCollection
   for (unsigned int i=0; i < deviceCollection.size(); ++i){
      PulsedMagnetDevice * pPMDev = deviceCollection[i];
      //do some processing
      ...

      //Then iterate on the PowerConverter device collection
      for (unsigned int j=0; j < pPowerConverterDevCol->size(); ++j{
         PowerConverterIntf::PowerConverterIntfDevice * pPCDev = (*pPowerConverterDevCol)[j];
         ...
```

PowerConverter deviceCollection reference initialization

Iteration on the PowerConverter deviceCollection

NotifyAction.cpp : we encourage in the design to define a specific RTAction "Notify", scheduled as any other RTAction which groups all notification messages. In our case notification message for PulsedMagnet can be automatic but we have manually to notify all the surrounding-classes.

```
Notify::Notify(const string& name, AbstractRTAction::RTActionConfig& rtActCfg) :
        RTAction<RTEvent, PulsedMagnetGlobalStore, PulsedMagnetDevice>(name, rtActCfg){

   // NotificationString is formatted:
   // "prop1,prop2,prop3:dev1,dev2,dev3,dev4&prop1,prop2,prop5:dev5"
   // where
   // separator ":" is used to separate property list from the device list
   // separator "&" is used to separate packets of properties/devices
   // separator "," is used to separate each strings
   // ATTENTION: you have to respect strictly the format because for the time being
   // it's not protected against white space or illegal format,
   // because it has been defined originally for private usage.

   // In this example we assume that the notification string is always the same
   // so the class contains three notification strings initialized once
   // Init of the notification string for PowerConverterIntf
   powerConverterNotifyStr = "Setting,Status,Acquisition:";
   vector<PowerConverterIntfDevice*>* pPCDevCol =
           PowerConverterIntf::PowerConverterIntfDevice::getDeviceCollection();
   for (unsigned int j=0; j < pPCDevCol->size(); j++) {
      Device* pDev = (*pPCDevCol)[j];
      powerConverterNotifyStr += pDev->name.get();
      if (j !=  pPCDevCol->size()-1)
         powerConverterNotifyStr += ",";
   }

   // Init the notification string for TimingIntf
   timingNotifyStr = "Setting,Status,Acquisition:";
   vector<TimingIntfDevice*>* pTimingDevCol =
           TimingIntf::TimingIntfDevice::getDeviceCollection();
   etc...
}
```

Prepare notification string for PowerConverterIntf Fesa class

```
Notify::execute(RTEvent* pEvent) {

   MultiplexingContext* pCtxt = pEvent->getMultiplexingContext();
   // Notify PowerConverterIntf
   string classStr =
       PowerConverterIntf::PowerConverterIntfDevice::pGlobalStore->name.get();
   AbstractEquipmentRT::notify(pCtxt, classStr, powerConverterNotifyStr);

   // Notify TimingIntf
   classStr = TimingIntf::TimingIntfDevice::pGlobalStore->name.get();
   AbstractEquipmentRT::notify(pCtxt, classStr, timingNotifyStr);
```

Notify PowerConverterIntf Fesa class

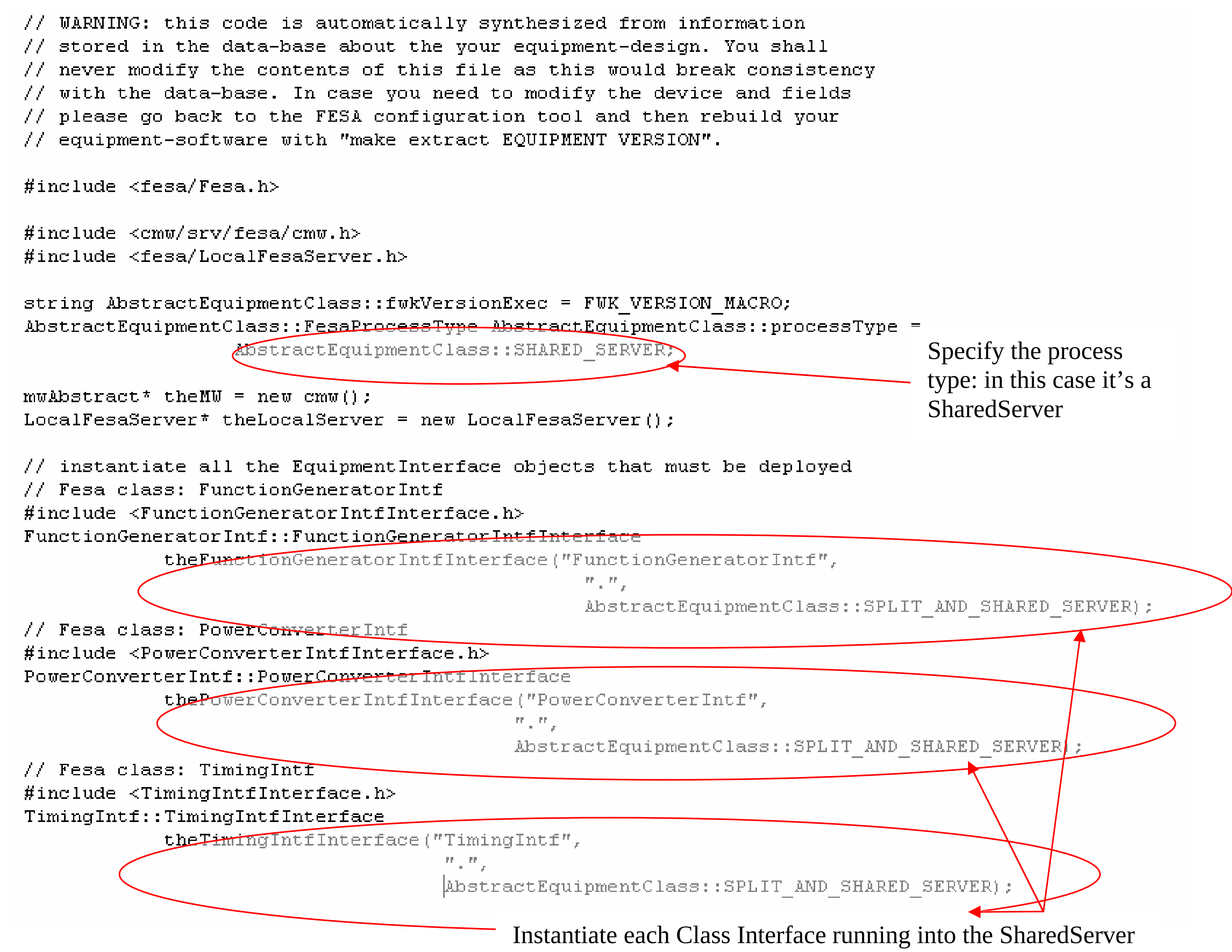
2.5 How to build a SharedServer and a separate RT process

Build those processes under TEST directory, can't be handled automatically and therefore requires some manual operations. You have to create two sources files:

- FesaSharedServer.cpp: which contains the declaration of all the Fesa classes deployed into the shared server
- RT4SS.cpp: means **R**eal**T**ime part for Shared**S**erver, which contains the declaration of the real time part of the Fesa class

2.5.1 Create FesaSharedServer.cpp source file

You can find here the FesaSharedServer.cpp corresponding to our use case:



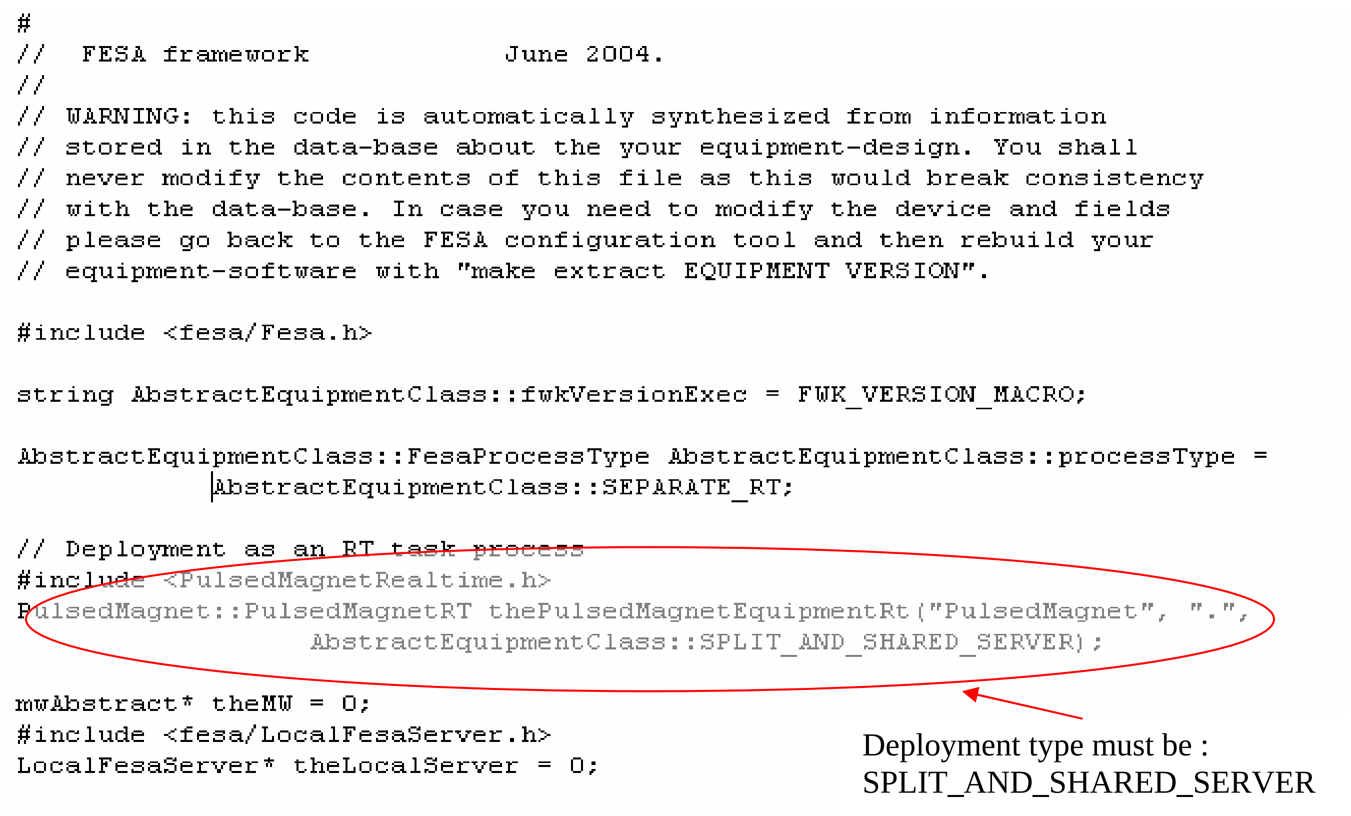
```
// WARNING: this code is automatically synthesized from information
// stored in the data-base about the your equipment-design. You shall
// never modify the contents of this file as this would break consistency
// with the data-base. In case you need to modify the device and fields
// please go back to the FESA configuration tool and then rebuild your
// equipment-software with "make extract EQUIPMENT VERSION".

#include <fesa/Fesa.h>

#include <cmw/srv/fesa/cmw.h>
#include <fesa/LocalFesaServer.h>

string AbstractEquipmentClass::fwkVersionExec = FWK_VERSION_MACRO;
AbstractEquipmentClass::FesaProcessType AbstractEquipmentClass::processType =
                 AbstractEquipmentClass::SHARED_SERVER;

mwAbstract* theMW = new cmw();
LocalFesaServer* theLocalServer = new LocalFesaServer();

// instantiate all the EquipmentInterface objects that must be deployed
// Fesa class: FunctionGeneratorIntf
#include <FunctionGeneratorIntfInterface.h>
FunctionGeneratorIntf::FunctionGeneratorIntfInterface
           theFunctionGeneratorIntfInterface("FunctionGeneratorIntf",
                                             ".",
                                             AbstractEquipmentClass::SPLIT_AND_SHARED_SERVER);
// Fesa class: PowerConverterIntf
#include <PowerConverterIntfInterface.h>
PowerConverterIntf::PowerConverterIntfInterface
           thePowerConverterIntfInterface("PowerConverterIntf",
                                        ".",
                                        AbstractEquipmentClass::SPLIT_AND_SHARED_SERVER);
// Fesa class: TimingIntf
#include <TimingIntfInterface.h>
TimingIntf::TimingIntfInterface
           theTimingIntfInterface("TimingIntf",
                                  ".",
                                  AbstractEquipmentClass::SPLIT_AND_SHARED_SERVER);
```

Specify the process type: in this case it's a SharedServer

Instantiate each Class Interface running into the SharedServer by specifying:
- "TimingIntf" : Fesa class name
- "." : path to retrieve instantiation document
- "SPLIT_AND_SHARED_SERVER" : deployment type

2.5.2 Create RT4SS.cpp

You can find here the RT4SS.cpp corresponding to our use case:



```
#
//  FESA framework              June 2004.
//
// WARNING: this code is automatically synthesized from information
// stored in the data-base about the your equipment-design. You shall
// never modify the contents of this file as this would break consistency
// with the data-base. In case you need to modify the device and fields
// please go back to the FESA configuration tool and then rebuild your
// equipment-software with "make extract EQUIPMENT VERSION".

#include <fesa/Fesa.h>

string AbstractEquipmentClass::fwkVersionExec = FWK_VERSION_MACRO;

AbstractEquipmentClass::FesaProcessType AbstractEquipmentClass::processType =
            AbstractEquipmentClass::SEPARATE_RT;

// Deployment as an RT task process
#include <PulsedMagnetRealtime.h>
PulsedMagnet::PulsedMagnetRT thePulsedMagnetEquipmentRt("PulsedMagnet", ".",
                  AbstractEquipmentClass::SPLIT_AND_SHARED_SERVER);

mwAbstract* theMW = 0;
#include <fesa/LocalFesaServer.h>
LocalFesaServer* theLocalServer = 0;
```

Deployment type must be :
SPLIT_AND_SHARED_SERVER

2.5.3 Update the Make.specific

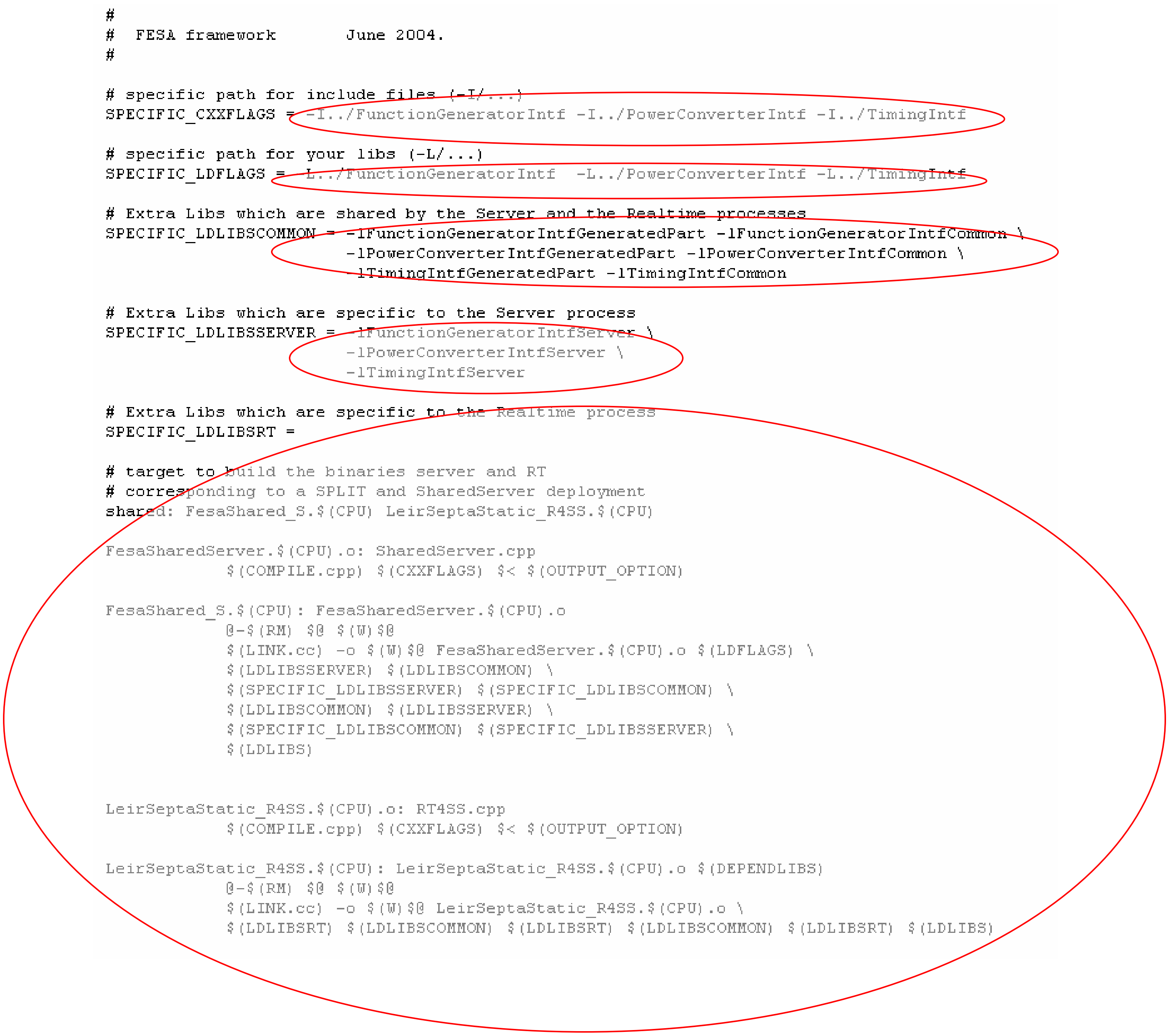
In order to build a SharedServer and the Realtime processes, you have to add a new target in the Makefile. Since all the Makefiles are re-generated each time a Fesa Synchronize is executed, this new target must be included into the Make.specific. This target is called “shared”.
So, to build the binaries execute :

```
> make CPU=x86 shared
```

The modifications consist to :

- Add all the necessaries include paths
- Add all the necessaries lib path
- Add the new target “shared” and the rules to build the expected binaries

You can find, below, the Make.specific corresponding to our use-case



```
#
#  FESA framework       June 2004.
#

# specific path for include files (-I/...)
SPECIFIC_CXXFLAGS = -I../FunctionGeneratorIntf -I../PowerConverterIntf -I../TimingIntf

# specific path for your libs (-L/...)
SPECIFIC_LDFLAGS = -L../FunctionGeneratorIntf  -L../PowerConverterIntf -L../TimingIntf

# Extra Libs which are shared by the Server and the Realtime processes
SPECIFIC_LDLIBSCOMMON = -lFunctionGeneratorIntfGeneratedPart -lFunctionGeneratorIntfCommon \
                        -lPowerConverterIntfGeneratedPart -lPowerConverterIntfCommon \
                        -lTimingIntfGeneratedPart -lTimingIntfCommon

# Extra Libs which are specific to the Server process
SPECIFIC_LDLIBSSERVER = -lFunctionGeneratorIntfServer \
                        -lPowerConverterIntfServer \
                        -lTimingIntfServer

# Extra Libs which are specific to the Realtime process
SPECIFIC_LDLIBSRT =

# target to build the binaries server and RT
# corresponding to a SPLIT and SharedServer deployment
shared: FesaShared_S.$(CPU) LeirSeptaStatic_R4SS.$(CPU)

FesaSharedServer.$(CPU).o: SharedServer.cpp
            $(COMPILE.cpp) $(CXXFLAGS) $< $(OUTPUT_OPTION)

FesaShared_S.$(CPU): FesaSharedServer.$(CPU).o
            @-$(RM) $@ $(W)$@
            $(LINK.cc) -o $(W)$@ FesaSharedServer.$(CPU).o $(LDFLAGS) \
            $(LDLIBSSERVER) $(LDLIBSCOMMON) \
            $(SPECIFIC_LDLIBSSERVER) $(SPECIFIC_LDLIBSCOMMON) \
            $(LDLIBSCOMMON) $(LDLIBSSERVER) \
            $(SPECIFIC_LDLIBSCOMMON) $(SPECIFIC_LDLIBSSERVER) \
            $(LDLIBS)


LeirSeptaStatic_R4SS.$(CPU).o: RT4SS.cpp
            $(COMPILE.cpp) $(CXXFLAGS) $< $(OUTPUT_OPTION)

LeirSeptaStatic_R4SS.$(CPU): LeirSeptaStatic_R4SS.$(CPU).o $(DEPENDLIBS)
            @-$(RM) $@ $(W)$@
            $(LINK.cc) -o $(W)$@ LeirSeptaStatic_R4SS.$(CPU).o \
            $(LDLIBSRT) $(LDLIBSCOMMON) $(LDLIBSRT) $(LDLIBSCOMMON) $(LDLIBSRT) $(LDLIBS)
```

2.5.4 Remarks

Testing deployment in mode split implies that all deviceModel of the involved Fesa classes are instantiated into Shared Memories. So during development and test phase, you will have several iterations, involving sometimes modification of the design or of the device instantiation document. In those cases, before you re-launch the new binaries, you have to delete the existing equipment's Shared Memory. For this you can either reboot your FEC or remove manually the equipment's Shared Memory by :

- LynxOS: using lipcs and lipcrm utility programs.
- Linux: removing the corresponding files located under /dev/shm

3 RDA Equipment Link

Some complex equipment of the accelerator, like Radio Frequency equipment, needs to be distributed on several computers (Front-End, PLC etc…). Application level would like to talk to a kind of “Virtual Equipment” which hides the complexity of the implementation.
So in this case we need to describe a distributed scheduler over several computers. For the time being Fesa doesn’t bring any specific support for this kind of complex equipment, but just recommend the usage of the CMW communication layer (called RDA) and in particular the subscription functionality and gives some advises to show how to integrate into a Fesa class such Equipment Link.

3.1 Use cases

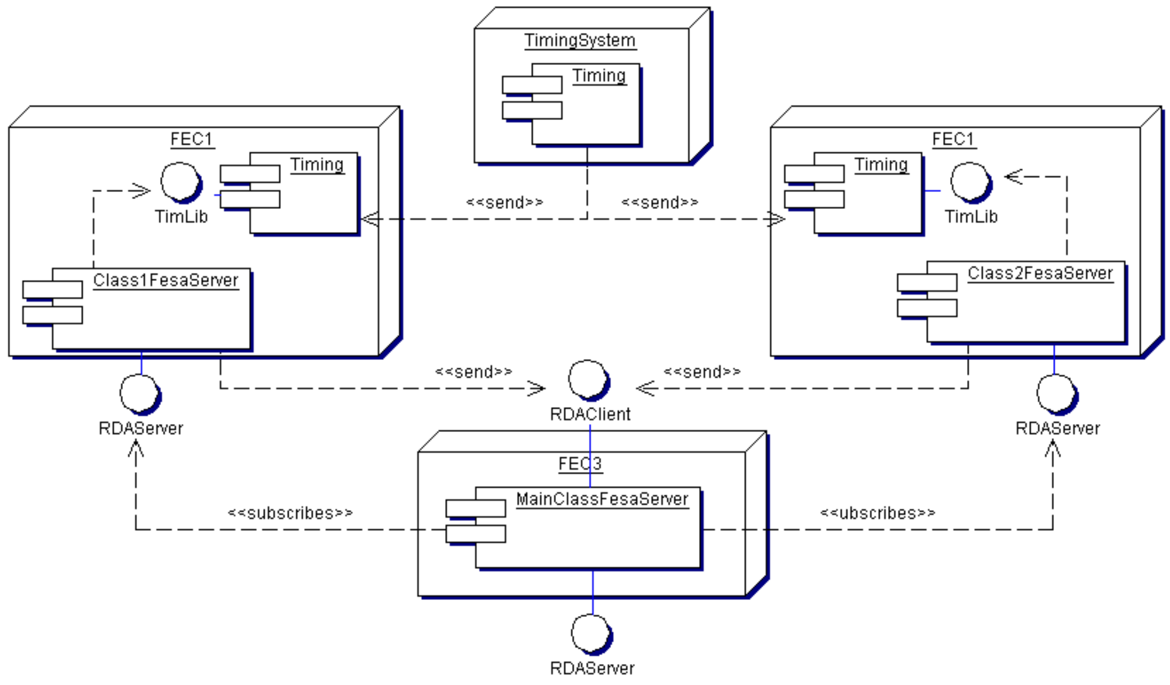
Let’s say a complex system deployed on three FECs. This system is described by three Fesa classes. Two of them are completely independent and in our example we call them Class1 and Class2, and the third one acts as the Main class and provide the virtual view of the complex equipment

3.2 Design

In this case all the classes are developed independently and you have not to registered in the node “equipment-link” any particular information

3.3 Deployment

From the Deployment Fesa Tool each class are deployed independently on separate Front-Ends, and the deployment diagram of our example is depicted in the following figure



3.4Development

3.4.1 Directories hierarchy

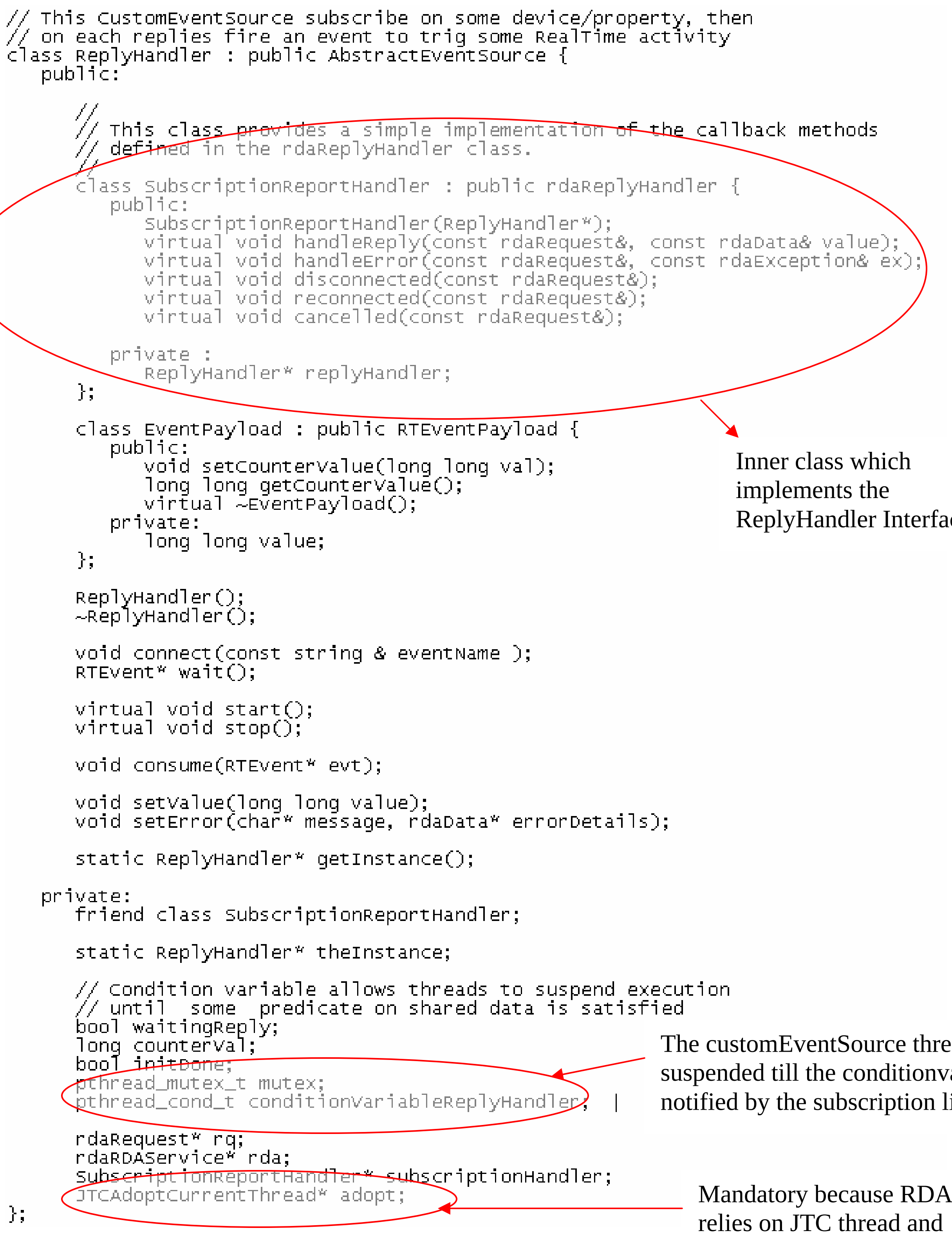
Each class are developed independently and since there is no library call from one class to an other there is nothing particular to set for the source development. Simply execute standard Fesa setup:
Fesa Setup Class1 0<scratch/edit>
Fesa Setup Class2 3 <scratch/edit>
Fesa Setup MainClass 1 <scratch/edit>

3.4.2 Source development

For Class1 and Class2 there is no particular recommendation.
In other hand, for the MainClass, we want to recommend a way to implement this RDA Link to avoid RDA connection spread everywhere in the source.
From the point of view of the MainClass, one of the EventSource which trigs the activity is the listening of the subscriptions. So we recommand to define a customEventSource which should contain all the mechanisms involved in subscription process:

- create replies Handlers
- start/stop subscription
- process the subscription replies, build a dedicated event payload, and fire an event to trig the appropriate RTAction

An example can be found in the Fesa class CVS repository: TestEqpLinksRDA version 0. From the package RT you can find an example of such customEventSource which is called ReplyHandler.cpp .
The following figure show just the header file of this customEventSource highlighting some important points.



```
// This CustomEventSource subscribe on some device/property, then
// on each replies fire an event to trig some RealTime activity
class ReplyHandler : public AbstractEventSource {
   public:

      //
      // This class provides a simple implementation of the callback methods
      // defined in the rdaReplyHandler class.
      //
      class SubscriptionReportHandler : public rdaReplyHandler {
         public:
            SubscriptionReportHandler(ReplyHandler*);
            virtual void handleReply(const rdaRequest&, const rdaData& value);
            virtual void handleError(const rdaRequest&, const rdaException& ex);
            virtual void disconnected(const rdaRequest&);
            virtual void reconnected(const rdaRequest&);
            virtual void cancelled(const rdaRequest&);

         private :
            ReplyHandler* replyHandler;
      };

      class EventPayload : public RTEventPayload {
         public:
            void setCounterValue(long long val);
            long long getCounterValue();
            virtual ~EventPayload();
         private:
            long long value;
      };

      ReplyHandler();
      ~ReplyHandler();

      void connect(const string & eventName );
      RTEvent* wait();

      virtual void start();
      virtual void stop();

      void consume(RTEvent* evt);

      void setValue(long long value);
      void setError(char* message, rdaData* errorDetails);

      static ReplyHandler* getInstance();

   private:
      friend class SubscriptionReportHandler;

      static ReplyHandler* theInstance;

      // Condition variable allows threads to suspend execution
      // until  some  predicate on shared data is satisfied
      bool waitingReply;
      long counterVal;
      bool initDone;
      pthread_mutex_t mutex;
      pthread_cond_t conditionVariableReplyHandler;   |

      rdaRequest* rq;
      rdaRDAService* rda;
      SubscriptionReportHandler* subscriptionHandler;
      JTCAdoptCurrentThread* adopt;
};
```

Inner class which implements the ReplyHandler Interface

The customEventSource thread is suspended till the conditionvariable is notified by the subscription listener

Mandatory because RDA relies on JTC thread and FESA Fwk use posix Thread

4 Interface equipment link

This relationship is used to talk with a Fesa class resident in the same Front-End. The main difference compare to the “RDA Link” is :

- It is restricted to the scope of the Front-End
- No communication is involved, it’s a function call

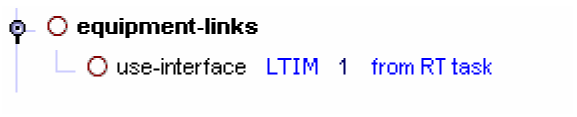
As for the “RDA Link”, there is, for the time being, no type checking at the compilation time.

4.1 Use case

A typical case is a Fesa class, for instance BPMLE which wants to control a Timing device through the LTIM Fesa class.

4.2 Design

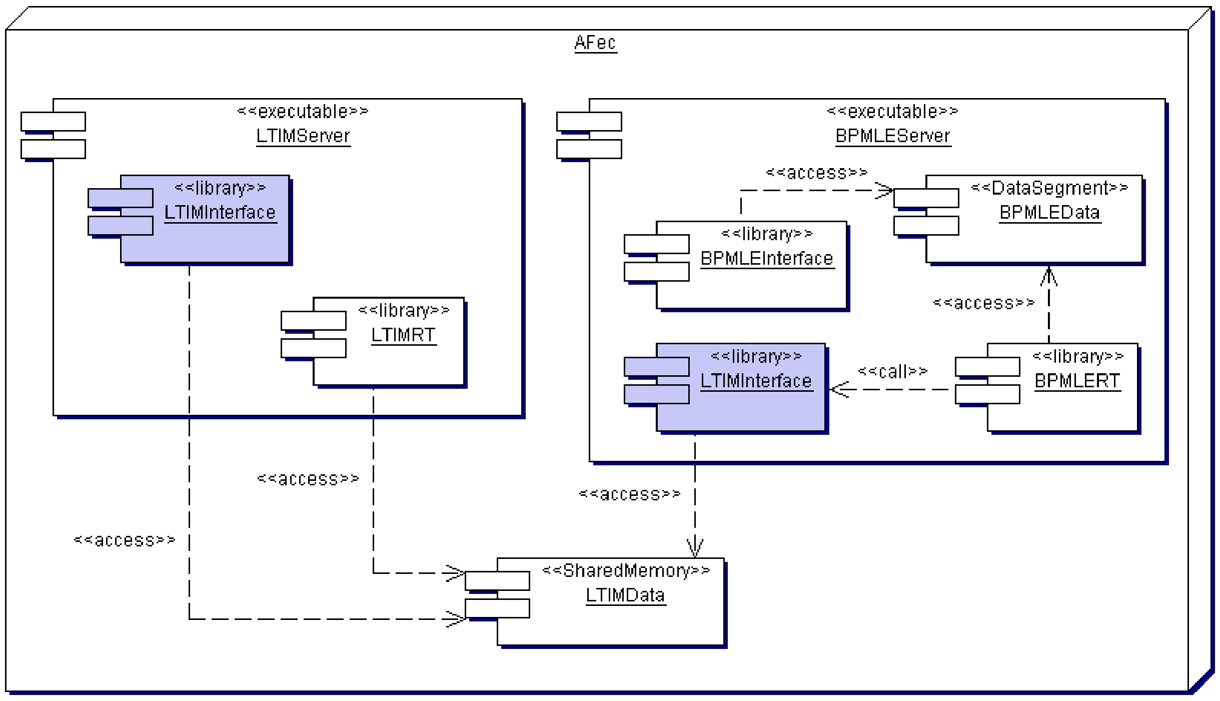
Such equipment link requires information in the design



equipment-links
use-interface LTIM 1 from RT task

4.3 Deployment

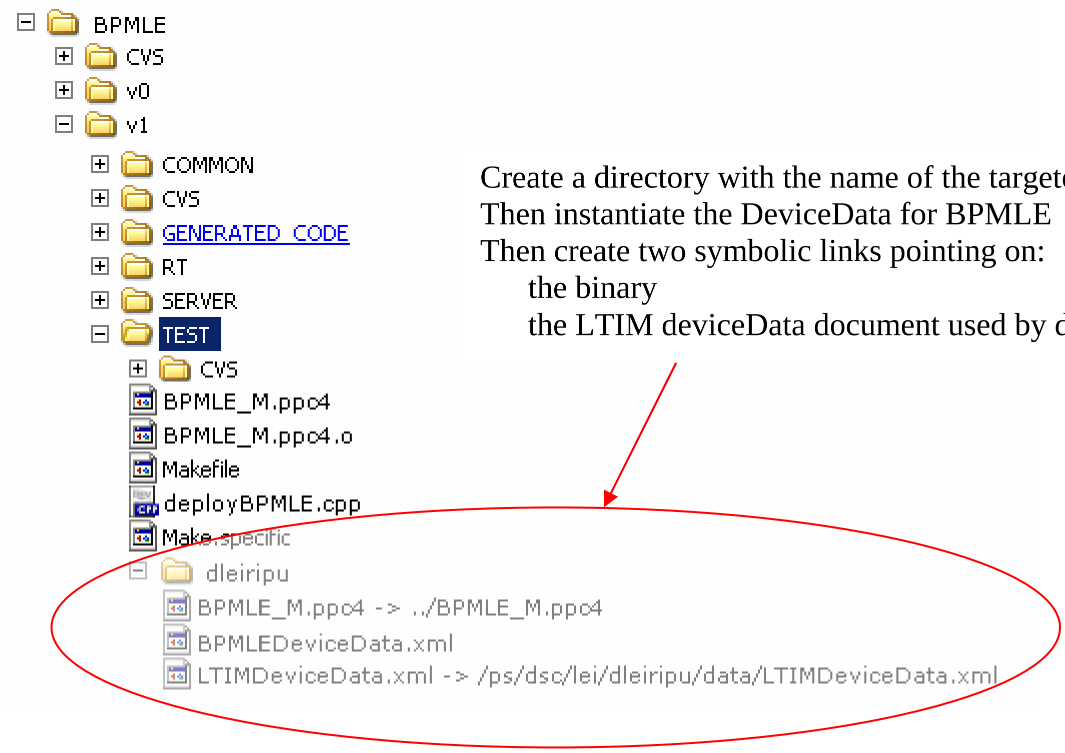
The deployment diagram for our use case is the following: as you can see the LTIMInterface library is duplicated into BPMLEI process



4.4 Development

4.4.1 Directories hierarchy

Since you use the interface of a deployed class, you have not to setup any particular directories hierarchy. The only think that you have to do, if you want to run the binary created in your TEST source directory is:
Let’s say that you want to test your fresh executable on the Front-end where your timing instance is deployed for instance, “dleiripu”. So we recommend to structure your TEST directory in a such way:

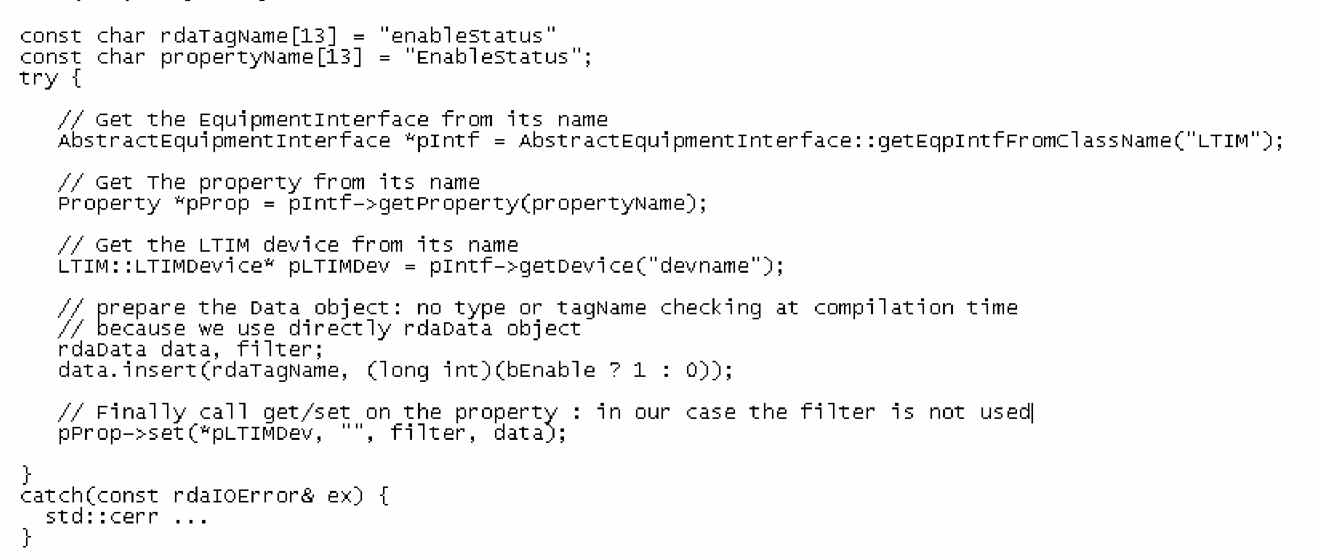


```
BPMLE
  CVS
  v0
  v1
    COMMON
    CVS
    GENERATED_CODE
    RT
    SERVER
    TEST
      CVS
      BPMLE_M.ppc4
      BPMLE_M.ppc4.o
      Makefile
      deployBPMLE.cpp
      Make.specific
      dleiripu
        BPMLE_M.ppc4 -> ../BPMLE_M.ppc4
        BPMLEDeviceData.xml
        LTIMDeviceData.xml -> /ps/dsc/lei/dleiripu/data/LTIMDeviceData.xml
```

Create a directory with the name of the targeted FEC
Then instantiate the DeviceData for BPMLE
Then create two symbolic links pointing on:
  the binary
  the LTIM deviceData document used by dleiripu

4.4.2 Source development

The picture illustrates the typical code that you have to produce to call a property on your favourite class:



```
const char rdaTagName[13] = "enableStatus"
const char propertyName[13] = "EnableStatus";
try {

   // Get the EquipmentInterface from its name
   AbstractEquipmentInterface *pIntf = AbstractEquipmentInterface::getEqpIntfFromClassName("LTIM");

   // Get The property from its name
   Property *pProp = pIntf->getProperty(propertyName);

   // Get the LTIM device from its name
   LTIM::LTIMDevice* pLTIMDev = pIntf->getDevice("devname");

   // prepare the Data object: no type or tagName checking at compilation time
   // because we use directly rdaData object
   rdaData data, filter;
   data.insert(rdaTagName, (long int)(bEnable ? 1 : 0));

   // Finally call get/set on the property : in our case the filter is not used
   pProp->set(*pLTIMDev, "", filter, data);

}
catch(const rdaIOError& ex) {
  std::cerr ...
}
```