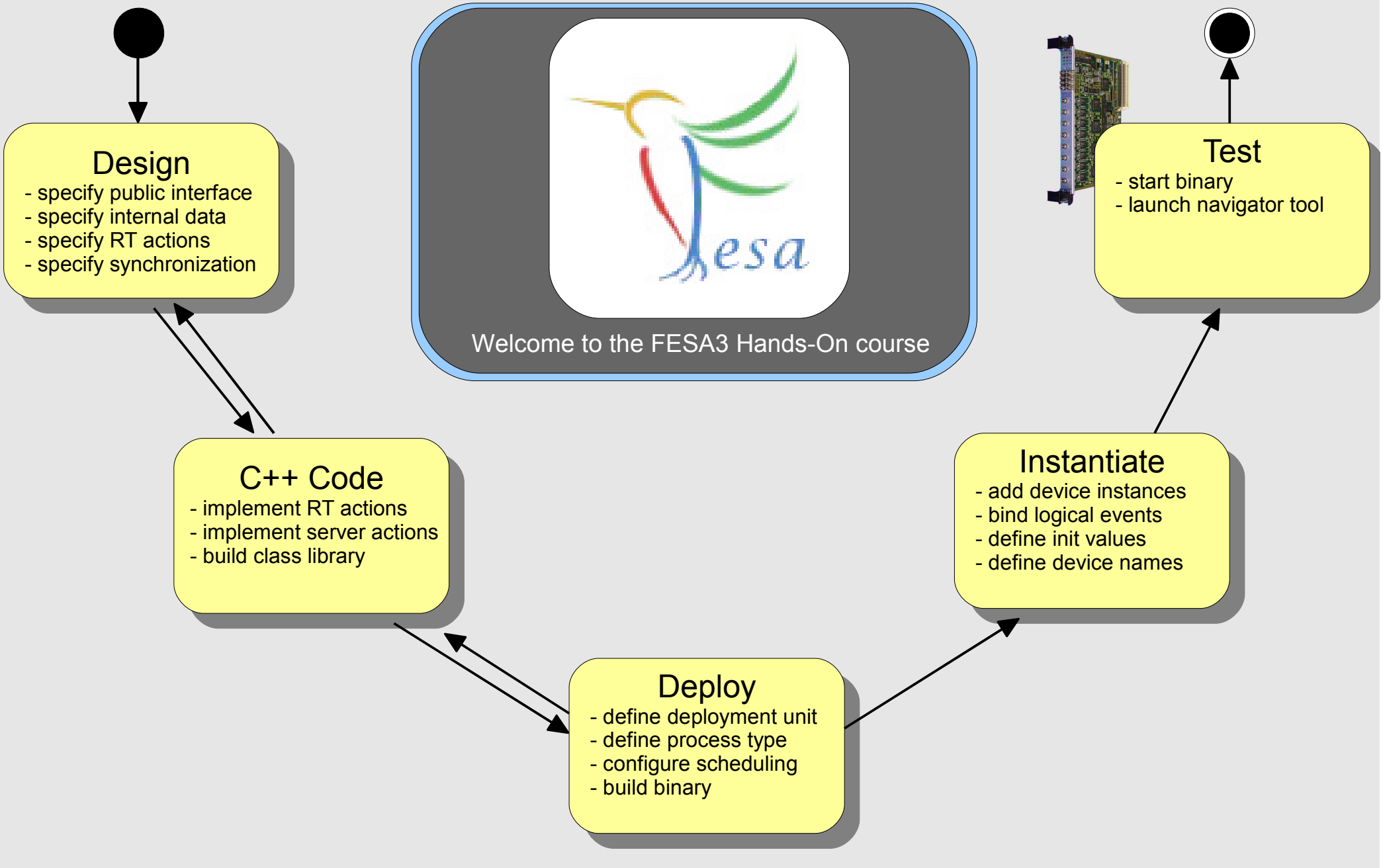
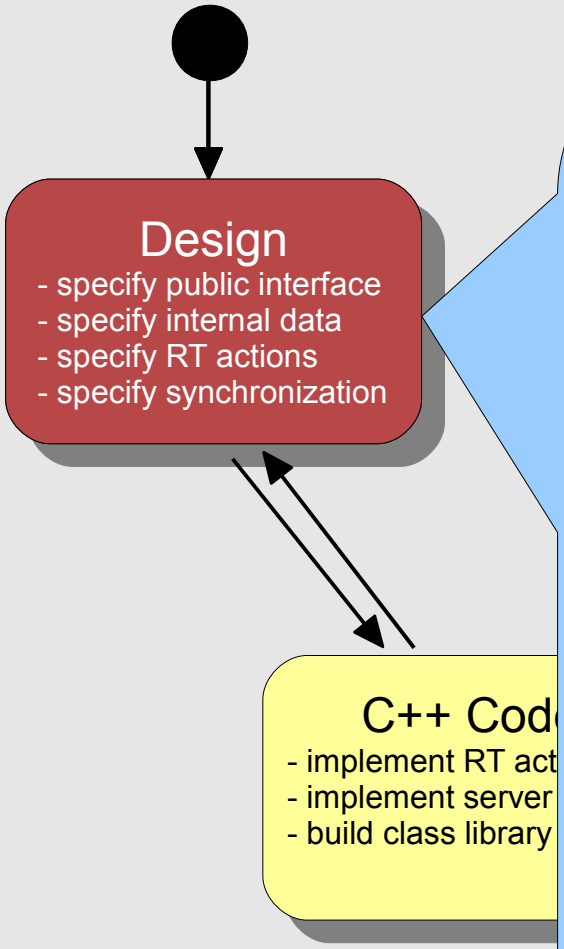
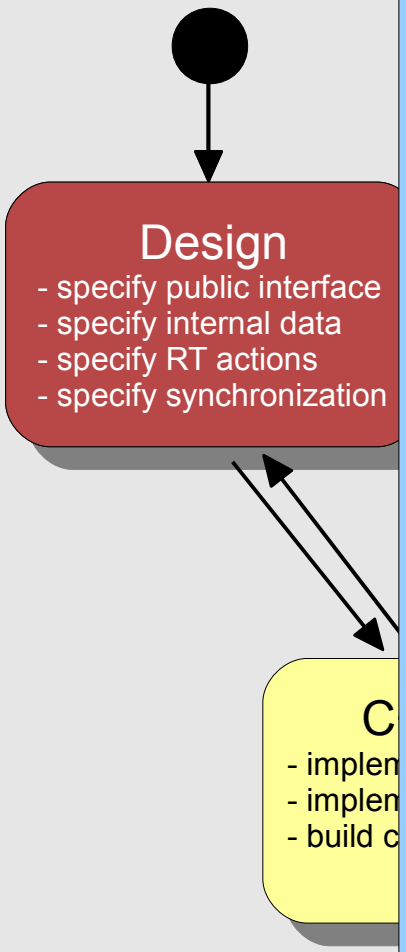

Welcome to the FESA3 Hands-On course
Design
- specify public interface
- specify internal data
- specify RT actions
- specify synchronization
C++ Code
- implement RT actions
- implement server actions
- build class library
Deploy
- define deployment unit
- define process type
- configure scheduling
- build binary
Instantiate
- add device instances
- bind logical events
- define init values
- define device names
Test
- start binary
- launch navigator tool



There are different ways to create a new FESA3 project within a new FESA3 class in Eclipse. The procedure below describes only one possible way. Anyhow, for your first steps with FESA3 it may help you to follow the predefined procedure:

1. Install the FESA3 Eclipse plug-in as described on the FESA3 web page
2. Start Eclipse
3. Open menu File → New → Project and choose FESA → New FESA project
4. Choose a unique name for your project. E.g. „YourNameTestProject_prj“
5. Choose Makefile Project → Empty Project as project type and Linux GCC as Tool chain
6. Press Finish to create your project
7. Right click on your project and choose FESA → Add Class
8. Again choose a unique name like „YourNameTestClass“
9. Choose the FESA-version you want to use and press Next
10. Choose the Empty template as class template and press Finish
11. Make sure the eclipse-view FESA browser is enabled.
    If it is not, you can enable it in the menu:
    Window → Show View → other → other → FESA browser
12. Create a design by adding elements as shown on the screen-shot on the next page. Note that the screen-shot only shows the elements you need to modify! Devote sufficient time for you to understand the meaning of each element appearing in the model's tree.
13. At each phase each file can be committed or retrieve from SVN. For more information about that visit the FESA3 web page.



Create your class-design as shown here

interface	(device-interface?, glo
device-interface	(setting?, acquisition?
acquisition	(acquisition-property?
acquisition-property	((description?), (filter-
name	Waveform
on-change	false
multiplexed	false
value-item	((description?, (scalar
name	waveformItem
array	((dim | custom-consta
type	double
custom-constant-dim	
constant-name-ref	WAVEFORM_SIZE
data-field-ref	
field-name-ref	waveform
get-action	(server-action-ref | abs
server-action-ref	
server-action-name-ref	Waveform
acquisition-property	((description?), (filter-
name	RectifiedWaveform
on-change	false
multiplexed	false
value-item	((description?, (scalar
name	waveformItem
array	((dim | custom-consta
type	double
custom-constant-dim	
constant-name-ref	WAVEFORM_SIZE
get-action	(server-action-ref | abs
server-action-ref	
server-action-name-ref	RectifyWaveform
custom-types	(notification-update-er
constant-unsigned-int	(description?)
name	WAVEFORM_SIZE
value	256
data	(device-data?, global-
device-data	(configuration?, settin
configuration	(hw-address?, device-
event-field	(description?)
name	tickField
setting	(state-field? | field? | (
field	((description?, (scalar
persistent	false
name	offset
multiplexed	false
scalar	
type	double
acquisition	(fault-field? | state-fiel
field	(description?, (scalar |
name	waveform
multiplexing	NONE
array	((dim | custom-consta
type	double
custom-constant-dim	
constant-name-ref	WAVEFORM_SIZE
field	(description?, (scalar |
name	phase
multiplexing	NONE
scalar	
type	double
actions	(get-server-action?, se
rt-action	((description?), (notifi
name	GenerateWaveform
notified-property	
property-name-ref	Waveform
notified-property	
property-name-ref	RectifiedWaveform
get-server-action	((description?), (disabl
implementation	custom
name	RectifyWaveform
get-server-action	((description?), (disabl
implementation	default
name	Waveform
events	(sources?, device-ever
sources	(timing-event-source?,
timer-event-source	(description?)
name	Timer
device-events	(logical-event+)
logical-event	((description?, (source
name	tick
source-ref	
source-name-ref	Timer
event-field-ref	
event-field-name-ref	tickField
scheduling-units	(scheduling-unit?)*
scheduling-unit	(selection-criterion?, r
name	TickSchedulingUnit
rt-action-ref	
rt-action-name-ref	GenerateWaveform
logical-event-ref	
logical-event-name-ref	tick

Validate your design with the validate button.

If your FESA-class design is valid you can trigger the code generation by pressing the button. This will generate the basic C++ source code of your class.

At the heart of your equipment's real-time activity, the generateWaveform real-time action class (from your design) is meant to be invoked each time a "tick"event occurs. You now need to enter the code of this action by filling-in the "GenerateWaveform::execute(...)" method in the src/RealTime/generateWaveform.cpp file.

The source code is as well stored in the FESA - SVN under the directory:/fesa-app/fesa-class/tutorials/MyNameTestProject_prj/MyNameTestClass

```
void GenerateWaveform::execute(fesa::RTEvent* pEvt)
{
    std::string logMessage;
    fesa::MultiplexingContext* pCtxt = pEvt->getMultiplexingContext();
    for(std::vector<Device*>::iterator device = deviceCol_.begin();device!=deviceCol_.end();device++)
    {
        try
        {
            double phase = (*device)->phase.get(pCtxt);
            double offset = (*device)->offset.get(pCtxt);
            for (unsigned int x=0; x<WAVEFORM_SIZE;x++)
                (*device)->waveform.setCell(sin(2*3.14*x/100.0+phase)+offset,x,pCtxt);
            (*device)->phase.set(phase+0.1,pCtxt);
            logMessage += "device:";
            logMessage += (*device)->getName();
            logMessage += "has been woken up by tick-event in order to generate a waveform.";
            pLog->send(logMessage.c_str(),pLog->traceType);
        }
        catch (fesa::FesaException& exception)
        {
            logMessage += "RTAction GenerateWaveform failed. Reason:";
            logMessage += exception.getMessage();
            pLog->send(logMessage.c_str(),pLog->traceType);
            throw;
        }
        catch(...)
        {
            logMessage += "RTAction GenerateWaveform failed for unknown reason.";
            pLog->send(logMessage.c_str(),pLog->traceType);
            throw;
        }
    }
}
```

C++ Code

- implement RT actions
- implement server actions
- build class library

- spe
- spe
- spe
- spe

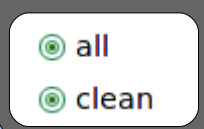
For the property „Waveform“ you don’t need to provide any piece of code, since in your design you specified a default get action. This means the get action for this property is automatically provided by the framework.

For the server action „RectifyWaveform“ of the property „RectifiedWaveform“ you selected „custom“ as implementation–type. This means you need to implement this action at your own. You can do so by filling the code below into the “RectifyWaveform::execute(...)” method in the src/Server/RectifyWaveform.cpp file.

```
void RectifyWaveform::execute(fesa::RequestEvent* pEvt,Device* pDev ,RectifiedWaveform_Data
{
    std::string logMessage;
    double recWaveForm[WAVEFORM_SIZE];
    unsigned long size = WAVEFORM_SIZE; //we cannot pass a const-type
    try
    {
        const double *pWaveform = pDev->waveform.get(size,pEvt->getMultiplexingContext());
        for(unsigned int x=0;x<WAVEFORM_SIZE;x++)
            recWaveForm[x]=fabs(pWaveform[x]);
        data.waveformItem.set(recWaveForm,WAVEFORM_SIZE);
        logMessage += "device:";
        logMessage += pDev->getName();
        logMessage += "has been remotely accessed.";
        pLog->send(logMessage.c_str(),pLog->traceType);
    }
    catch (fesa::FesaException& exception)
    {
        logMessage += "ServerAction RectifyWaveform failed. Reason:";
        logMessage += exception.getMessage();
        pLog->send(logMessage.c_str(),pLog->traceType);
        throw;
    }
    catch(...)
    {
        logMessage += "ServerAction RectifyWaveform failed for unknown reason.";
        pLog->send(logMessage.c_str(),pLog->traceType);
        throw;
    }
}
```

C++ Code

- implement RT actions
- implement server actions
- build class library



After you finished the implementation of both actions, you can build your FESA-class library by going to the folder /src and executing „make all“. This can be done either in Eclipse using the „Make Targets“ view in the C++ perspective, or directly from the Linux-console.

By executing „make clean“ you can remove all older libraries and object files.

In FESA3 the binary which is launched on a front-end may consist of a mix of different classes (for example when using the composition or the inheritance concept). To define these classes and their relations to each other a „deploy unit“ is used. To add a deploy unit, open your class design and push the “add deploy unit” button.

Choose an unique name, like „YourNameDeployUnit“ and take a look at the generated deployment document.

class	
class-name	YourNameTestClass
scheduler	(concurrency-layer?)+
concurrency-layer	(scheduling-unit)+
name	tickLayer
event-queue-size	10
scheduling-unit	
scheduling-unit-name-ref	YourNameTestClass::TickSchedulingUnit
executable	((rt | server | mixed))+
mixed	
extension	_M

Again, only the items that you need to add or change are listed At the screen-shot at the left side.

If you finished editing the deployment document, validate the document and generate the C++ source code by using the buttons and .

To obtain an executable FESA binary-file, you need compile the generated deploy unit code and link it together with the class library and the framework libraries. This will be done if you enter the deploy unit folder and execute „make all“. This possible using Eclipse or the Linux-console.

Deploy

- define deployment unit
- define process type
- configure scheduling
- build binary

For the next step you need to configure on which frontend Your binary should run. To do so, re-open your class design and push the „Add FEC“ button and put in the name of the frontend on which you currently work.

events-mapping	(device-events)
device-events	(tick)
tick	(Timer)
Timer	(concrete-event+)
concrete-event	(period)
value	YourNameTestClass::tick
period	
value	1000
device-instance	(configuration)
name	MyNameDevice1
configuration	(description, accelerator, ti
accelerator	
value	NONE
timingDomain	
value	NONE
tickField	
value	YourNameTestClass::tick
device-instance	(configuration)
name	MyNameDevice2
configuration	(description, accelerator, ti
accelerator	
value	NONE
timingDomain	
value	NONE
tickField	
value	YourNameTestClass::tick
global-instance	(configuration)
accelerator	
value	NONE
timingDomain	
value	NONE

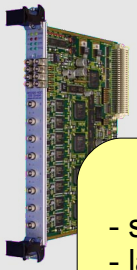
After that press to create a new instance of your class for this frontend.

The instantiation document of your front-end will now open automatically. You just need to configure the devices of your class, as described on the screen-shot on the left.

As always, you can validate your instantiation document by pressing .

rse

- build binary



Test
- start binary
- launch navigator tool

Instantiate
- add device instances
- bind logical events
- define init values
- define device names

To test your deploy unit binary, you need to open a Linux-console, browse to the folder of your instantiation file and start your deploy unit binary from there.
Here a screen-shot how this looks on the Linux-console of my system:

```
# cd workspace/MyNameTestProject_prj/MyNameTestClass/v0/src/Test/asl724/
# ../MyNameTestDeployUnit/MyNameTestDeployUnit_M0.0.L865
```

Congratulations! You can now remotely access the devices “MyNameDevice1” and “MyNameDevice2” across the middleware. One client for this purpose is the FESA-Navigator. In Eclipse you can start the Navigator by pressing when you are at the instantiation document.

Des
- specify pul
- specify inte
- specify RT
- specify syn

Test
- start binary
- launch navigator tool

C++ Code
- implement RT actions
- implement server actions
- build class library

Instantiate
- add device instances
- bind logical events
- define init values
- define device names

Deploy
- define deployment unit
- define process type
- configure scheduling
- build binary