



# **Developers Guide**

*Release 0.9beta2*

**Daniel Bültmann et al., Department of Communication  
Networks (ComNets), RWTH Aachen University**

February 08, 2012



# CONTENTS

<b>1</b>	<b>About this Book</b>	<b>1</b>
1.1	How to create this Book from the Sources . . . . .	1
<b>2</b>	<b>Getting Started</b>	<b>3</b>
2.1	Prerequisites . . . . .	3
2.2	Download . . . . .	4
2.3	Installation . . . . .	5
2.4	SDKLayout . . . . .	6
2.5	Installing the Wrowser . . . . .	7
2.6	Installing the PostgreSQL database . . . . .	9
2.7	Cygwin . . . . .	11
<b>3</b>	<b>The openWNS Software Developer's Kit (SDK)</b>	<b>13</b>
3.1	Mastering your SDK (a guide to playground.py) . . . . .	13
3.2	Creating and Maintaining Development Branches . . . . .	16
3.3	Performance Tips . . . . .	17
<b>4</b>	<b>The Simulation Platform</b>	<b>19</b>
4.1	The Event Scheduler . . . . .	19
4.2	Random Number Distributions . . . . .	22
4.3	Measurement Sources . . . . .	26
4.4	Context Collector . . . . .	29
4.5	Evaluation Framework . . . . .	34
4.6	Smart Pointers . . . . .	39
4.7	How to write a Finite State Machine . . . . .	40
<b>5</b>	<b>Writing Code</b>	<b>49</b>
5.1	Writing Documentation . . . . .	49
5.2	Writing C++ Unit Tests . . . . .	51
5.3	Writing Functional Units Tests . . . . .	54
5.4	Writing Python Unit Tests . . . . .	58
<b>6</b>	<b>Coding Guidelines</b>	<b>61</b>
6.1	Why do we need a style guide for openWNS? . . . . .	61
6.2	Code layout . . . . .	61
6.3	Naming Conventions . . . . .	65
6.4	Programming recommendations . . . . .	68
6.5	License Statement . . . . .	69
<b>7</b>	<b>Coding Guidelines Reference Card</b>	<b>71</b>

7.1	Layout Rules . . . . .	71
7.2	Naming Conventions . . . . .	71
<b>8</b>	<b>Debugging your Code</b>	<b>73</b>
8.1	Finding Memory leaks . . . . .	73
8.2	CPU Profiling . . . . .	73
8.3	SmartPtr Debugging . . . . .	73
<b>9</b>	<b>Todo List</b>	<b>75</b>
<b>10</b>	<b>Indices and tables</b>	<b>77</b>
	<b>Index</b>	<b>79</b>

# ABOUT THIS BOOK

This book is mainly generated by Doxygen. Documentation is written in Doxygen markup and most of the chapters in this book are extracted from Doxygen. The openWNS SDK provides helpers to define the structure of the book and re-order page sections. Doxygen has no builtin support for this.

**Note:** There is some bug in the interaction of Doxygen and Latex. In this book every piece of example code is ended with `@c \end{verbatim}` tag. Please simply ignore this line for now.

## 1.1 How to create this Book from the Sources

Within your openWNS SDK execute:

```
> ./playground.py createmanuals
```



# GETTING STARTED

## 2.1 Prerequisites

openWNS relies on a number of third party software. Each of the listed libraries and programs below are freely available. openWNS is built entirely around free software. Some of the software is optional.

### 2.1.1 Third party libraries

- **CppUnit** ( $\geq 1.10$ ) The basic unit testing framework
- **Python** ( $\geq 2.5$ ) Used by almost everything starting from the build framework to configuration
  - Numpy Python package used for channel models implemented in Python
- **Boost** Mainly used to have the TR1 implementations of the upcoming C++ standard available
  - Besides basic Boost files Boost libraries Signals, Filesystem, Date-Time and Program-Options are required

### 2.1.2 Build framework

- **Bazaar** The Revision Control System
- **SCons** ( $\geq 0.96$ ) A make replacement
- **GCC** ( $\geq 3.4$ ) Compiler Suite, due to usage of special language features regarding templates the versions below 3.4 do not work

### 2.1.3 Optional for build framework

- **Doxygen** Documentation is generated with this tool
- **Graphviz** Used by doxygen to create nice UML diagrams
- **Iccream** Allows for distributed compiling

### 2.1.4 Ubuntu Linux

Ubuntu Linux 10.04 is currently the default development and testing platform for openWNS. All required packages can be easily downloaded using the command:

```
$ sudo apt-get install build-essential scones libboost-dev libboost-program-options-dev libboost-date-
```

if you are using the previous Long Term Support (LTS) version 8.04 (Hardy) of Ubuntu Linux you have to type:

```
$ sudo apt-get install build-essential scones libboost-dev libboost-program-options-dev libboost-date-
```

You may encounter problems with Bazaar not being able to access the repository. Follow the instructions here to install the newest, not officially supported, version of Bazaar for Ubuntu Linux 8.04 (Hardy): <https://launchpad.net/~bzt/+archive/ppa>.

## 2.2 Download

**Important:** Before you actually start with the download make sure your system has all the necessary software to install and run openWNS: see *Prerequisites*.

**Note:** These download instructions are a little bit longer than others (e.g. where you just need to download a .tgz-file). But it pays off as soon as you want to retrieve your first update. In the end the download should not take more than 5 minutes (time for downloading not included).

### 2.2.1 Short version

```
$ bzt branch http://launchpad.net/~comnets/openwns-sdk/sdk--main--1.0 myOpenWNS
$ cd myOpenWNS
$ ./playground.py upgrade --noAsk
```

### 2.2.2 Configuring Bazaar

Like many other software projects openWNS is available via a revision control system. The system is called Bazaar. Thus, in order to download openWNS you need Bazaar. For most distributions (SuSE, Debian, Ubuntu, ...) Bazaar is available as package. In any other case you will need to build it from the sources available [here](#)

To see if Bazaar is installed on your system and to check that you have an up to date version you can try:

```
$ bzt --version
Bazaar (bzt) 0.92.0
...
```

If you haven't used Bazaar before you need to make yourself known to the system:

```
$ bzt whoami "Joe Average <joe@average.com>"
```

Your id should be your name, followed by your email address in angle brackets. Bazaar records your id in the log messages for your commits. This information will @em not be used at any point when you're just downloading openWNS (like now). For further information on Bazaar you can have a look at <http://bazaar-vcs.org>.

### 2.2.3 Retrieve a copy of openWNS

Now it's time to retrieve a copy of openWNS. The following command will checkout the current version of openWNS to 'myOpenWNS'. Choose whatever you want as name.

```
$ bazaar branch http://launchpad.net/~comnets/openwns-sdk/sdk--main--1.0 myOpenWNS
Branched 151 revision(s).
$ cd myOpenWNS
```

Now you have a local copy of openWNS. Well, not really. What you have is rather an empty house. If you inspect all the sub-directories of `myOpenWNS` at this moment, you would notice that they are almost all empty. Apart from some `bash` and `Python` scripts there is not much to see. Especially no `C++` source code below the directory `framework` or `modules`.

```
$ ls framework/
```

So let's furnish this house! `openWNS` is designed to be a highly modular simulation framework. Hence, it is made up of a number of modules. Each module again is a `Bazaar` project (just like the one you've just fetched). Normally you would have to fetch each `Bazaar` project (or each module) by hand (like you did with this). This is very tedious. Fortunately, there is a program that helps you with this task (and other task as you will learn). It is called `playground.py`. So to fetch all modules and necessary other data simply enter:

```
$ ./playground.py upgrade
Warning: According to 'config/projects.py' the following directories are missing:
[.. many directories ..]
Try fetch the according projects? (Y/n) y
```

Just answer `y` to this question and all necessary projects will be fetched. Depending on your link speed and the current size of `openWNS` this can take several minutes.

After the download has finished you have all pieces available to proceed with the installation of `openWNS`. Now there should be the framework available:

```
$ ls framework/*

application/
dllbase/
library/
pywns/
rise/
```

## 2.3 Installation

After you have successfully downloaded `openWNS` according to [Download](#) you are ready to install `openWNS`. Make sure you satisfy all prerequisites in [Prerequisites](#).

The installation itself is quite easy:

```
$ ./playground.py install --flavour=dbg
```

After building has finished you can find the respective modules in `./sandbox/dbg/`.

Under `./sandbox/dbg/bin/` you should find a binary which is called `openwns`.

### 2.3.1 Test the Installation

openWNS supports the testing of developed code using both unit and system tests. To test the installation, you can first run the unit tests. Change to the directory

```
$ cd tests/unit/unitTests
```

and say

```
$ ./openwns -t -v
Loading...
Library: ofdmaphy
Module: ofdmaphy
[...]
[TST] wns::fsm::tests::FSMTest::multipleStateCreations

OK (1052 tests)
```

[OK]

`-t` runs all available unit tests and `-v` puts `openwns` into verbose mode (`-help` shows all available options).

For a more extensive testing of all installed modules the system tests can be used. System tests are made of a configuration file of the openWNS which sets up a simple simulation scenario of few nodes, each one equipped with certain modules, e.g. with a Layer 2 according to IEEE 802.11. Additionally, a system test contains reference output which is compared with the actual output of the simulation.

To run all system tests, you first have to compile the `opt` version using `playground.py`:

```
$ ./playground.py install --flavour=opt
```

After this, you can start all system test with

```
$ ./playground.py runtests
Starting test suites ...
NOTE: you may see slow progress since the tests run simulations

*****
SystemTestSuite: ...x/source/cleanWNS/tests/system/ip-Tests--main--1.0
Configuration: config.py
Description: A ring of three subnets to test IP
-----
Preparation phase:
Running simulations (no test, just preparing output) in debugging and
optimized mode (may take very long):
[...]
Ran 1161 tests in 490.323s

OK
```

If one of the tests did not end up with OK, something went very wrong during the installation. Please check if all previous steps were successful.

## 2.4 SDKLayout

The SDK (Software Development Kit) keeps all other sub projects of openWNS. The structure of the SDK as well as the location of the sub projects is as follows (note, a directory followed by a name in square means the directory is a

sub project):

- `openWNS-sdk/`: master project
- `openWNS-sdk/bin/`: helper scripts
- `openWNS-sdk/config/`: configuration for the SDK
- `openWNS-sdk/config/projects.py`: defines the projects being part of this working copy
- `openWNS-sdk/config/private.py`: user defined compilation settings
- `openWNS-sdk/config/pushMailRecipients.py`: list of recipients for mail on 'bzt push'
- `openWNS-sdk/config/valgrind.supp`: openWNS-specific valgrind suppressions
- `openWNS-sdk/documentation/`: the documentation project
- `openWNS-sdk/framework/`: core part (lib, simulator application) of openWNS
- `openWNS-sdk/framework/application/`: core application project
- `openWNS-sdk/framework/library/`: core library project
- `openWNS-sdk/framework/rise/`: Layer 1 and channel library
- `openWNS-sdk/framework/dllbase/`: Layer 2 library
- `openWNS-sdk/framework/pywns/`: post processing, system tests in python
- `openWNS-sdk/modules/`: Modules for different entities in the ISO/OSI protocol stack
- `openWNS-sdk/sandbox/`: the build system will install libs and apps here
- `openWNS-sdk/tests/`
- `openWNS-sdk/tests/unit/`: Python and C++ unit tests
- `openWNS-sdk/tests/system/`: System tests
- `openWNS-sdk/wnsbase/`: SDK builtins

Most important are probably the `framework` and `tests/unit` directory ;-).

## 2.5 Installing the Wrowser

The Wrowser (an acronym for Wireless network simulator Result brOWSER) supports with a graphical interface the collection of results, extraction of measurements and creation of parameter plots. Furthermore, the Wrowser helps to create simulation campaigns with large parameter spaces. The Wrowser is a project separated from the openWNS and thus has to be installed independently.

**Important:** The following steps are only required if you want to use the Wrowse to view simulation results. To run a single simulation and to evaluate the results using the text-output files, the Wrowser is not needed.

## 2.5.1 Prerequisites

Similar to the openWNS the Wrowser relies third party software. Each of the listed libraries and programs below are freely available.

- `python-qt4` A comprehensive set of Python bindings for the Qt cross-platform GUI/XML/SQL C++ framework from Qt Software
- `python-qt4-dev` Tools Development tools for `python-qt4`
- `python-matplotlib` A python 2D plotting library which produces publication quality figures in a variety of hard-copy formats and interactive environments across platforms.
- `python-tk` Python's de-facto standard GUI (Graphical User Interface) package.
- `python-scipy` Python library for mathematics, science, and engineering
- `pyqt4-dev-tools` Various support tools for PyQt4 developers

Use the following command to install them in Ubuntu Linux:

```
$ sudo apt-get install python-qt4 python-qt4-dev python-matplotlib python-tk python-scipy pyqt4-dev-t
```

## 2.5.2 Installation

Similar to the openWNS installation, the Wrowser can be obtained and updated using Bazaar. The following command will checkout the current version of the Wrowser into the directory 'wrowser'. Choose whatever you want as name. Note that for the last step the root password is required.

```
$ bazaar branch lp:openwns-wrowser wrowser
Branched 27 revision(s).
$ cd wrowser
$ sudo python setup.py install
```

You should now be able to start the Wrowser using

```
$ wrowser
```

It will open an empty window, see *The initial, empty Wrowser window*.

## 2.5.3 Checking the playground.py wrowser-plugin

Go into your openWNS - directory created during the installation of the openWNS and call `playground.py`. If everything went right, you should see the new command `preparecampaign`:

```
$ ./playground.py
[...]
preparecampaign : Prepare a simulation campaign.
[...]
```

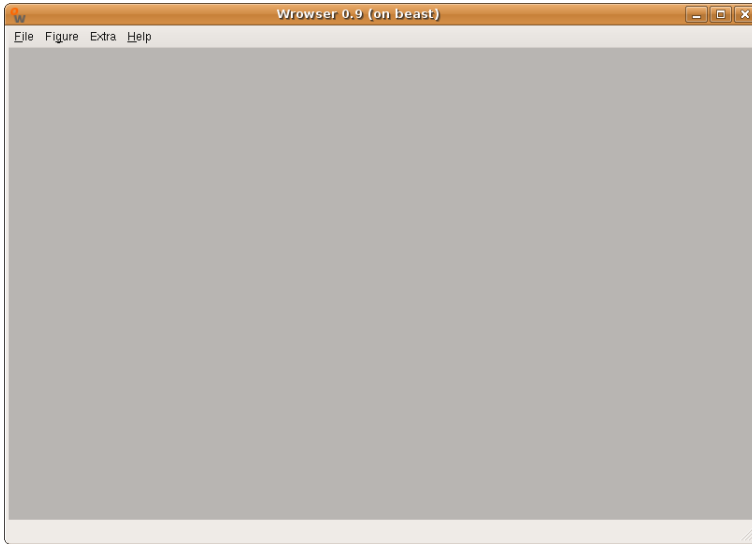


Figure 2.1: The initial, empty Wrowser window

## 2.6 Installing the PostgreSQL database

To allow for an efficient viewing, analyzing and management of large amounts of simulation output, the openWNS supports a database frontend based on the open-source PostgreSQL database. In the following, this database will be configured to work together with the openWNS.

**Important:** The following steps are only required if you want to store simulation results into a PostgreSQL database. To run a single simulation this is not required.

### 2.6.1 Prerequisites

- PostgreSQL A powerful, open source object-relational database system.
- python-psycopg2 (>= 2.0) A PostgreSQL database adapter for the Python programming language.

### 2.6.2 Adding a database user

The following section is only valid for a virgin installation of the PostgreSQL without any existing users. If you would like to use an already existing PostgreSQL, some of the following steps are not necessary. Please contact your database administrator and the PostgreSQL manual for further information.

1. After a virgin installation, the database-admin user `postgres` does not have any password. To change it to `foobar`, type

```
$ su postgres
$ psql
Welcome to psql 8.3.6, the PostgreSQL interactive terminal.

postgres=# ALTER ROLE postgres PASSWORD 'foobar';
ALTER ROLE
postgres=# \q
$ exit
```

- Furthermore, the database can be accessed from everywhere, not only locally. To change this, edit the file `/etc/postgresql/8.3/main/pg_hba.conf`:

- Comment out the line `local all postgres ident sameuser`.
- Change the line `local all all ident sameuser` to `local all all md5`.

Then, restart PostgreSQL with

```
$ /etc/init.d/postgresql-8.3 restart
```

- Now, you can create the database required for the simulation. The default name in all scripts is `simdb`, so it is recommended to use this name:

```
$ su postgres
$ psql
Password: foobar
Welcome to psql 8.3.6, the PostgreSQL interactive terminal.

postgres=# CREATE DATABASE simdb;
postgres=# \q
$ exit
```

- Then, the necessary tables to store simulation campaign parameters and results have to be created. For this purpose, a script can be used which is part of the `wrowser`. It can be found in the directory `wrowser/simdb/sql`:

```
$ cd wrowser/simdb/sql
$ psql -U postgres -d simdb -f setupSimDB.sql
```

- Finally, a database user account for your user account must be created. Again, a script is prepared for this, now in the directory `wrowser/simdb/scripts`:

```
$ cd wrowser/simdb/scripts
$ ./createUser.py
```

**Note:** Before running the `./createUser`, check that in this script the two variables `hostname` and `dbName` are set to the correct values, i.e., `localhost` and `simdb` if this installation guide is followed.

**Important:** The script assigns a default password to the created user account, which is `foobar`. You can change this password by editing the variable `password` in the script before running. Do not use your default user password - the database user account password is not stored secure!

Now, a account is created on the database which can be used to store the simulation results.

### 2.6.3 Configuring the database information in the Wrowser

The Wrowser needs to know where to find the campaign database. This can be configured by starting the Wrowser and then selecting `Extra, Preferences` in the menu. In the following dialog, see [Setting the database.](#), please fill in the following values:

- Hostname: `localhost`
- Databasename: `simdb`
- Username: Your user account name
- Password: `foobar` if not changed in the `createUser.py` script.

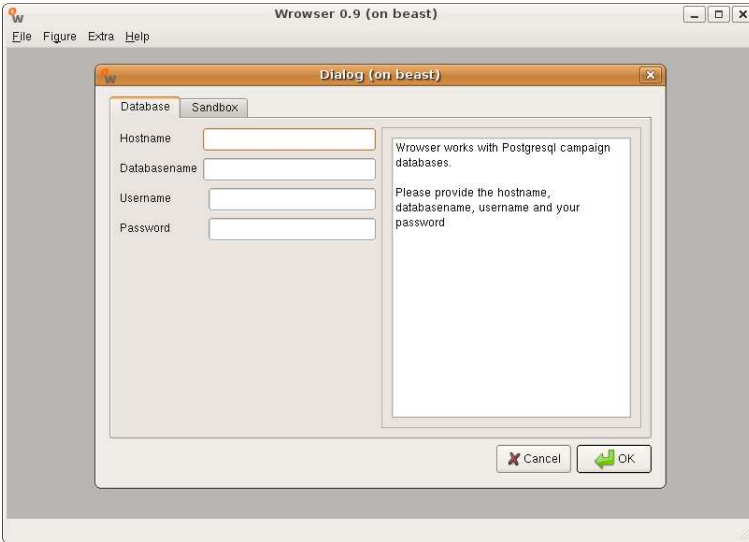


Figure 2.2: Setting the database.

Then, select the tab `Sandbox` and fill in the complete path to the sandbox of your openWNS installation, e.g., `/home/userName/myOpenWNS/sandbox`.

## 2.7 Cygwin

### 2.7.1 Installing Cygwin

Download Cygwin from [here](#). For this guide version 1.5.25-15 has been tested. A full installation of all Cygwin packets was done to assure dependencies are met.

### 2.7.2 Scons

There is no binary package of the build tool Scons for Cygwin so you need to download the Tarball from [here](#). Version 1.2.0 was successfully tested. Untar and unzip the downloaded file, enter the newly created directory and run

```
$ python setup.py install
```

### 2.7.3 Boost

Boost 1.33 libraries are included in tested Cygwin version. Unfortunately there is a bug in one file that needs to be patched. Alternatively Boost  $\geq 1.34$  could be installed. Get the patch from [here](#) or simply edit `/usr/include/boost-1_33_1/boost/numeric/ublas/lu.hpp`.

Create a symbolic link to the Boost header files:

```
$ ln -s /usr/include/boost-1_33_1/boost /usr/include/boost
```

Create symbolic links to the libraries:

```
$ ln -s /usr/lib/libboost_date_time-gcc-mt-s.a /usr/lib/libboost_date_time.a
$ ln -s /usr/lib/libboost_program_options-gcc-mt-s.a /usr/lib/libboost_program_options.a
$ ln -s /usr/lib/libboost_signals-gcc-mt-s.a /usr/lib/libboost_signals.a
```

### 2.7.4 Python

Create a symbolic link to the Python library:

```
$ ln -s /usr/lib/python2.5/config/libpython2.5.dll.a /usr/lib/libpython2.5.a
```

### 2.7.5 Compiling openWNS

You have to compile openWNS statically under Cygwin using by including the `--static` option to `playground.py install`.

```
$ ./playground.py install --static
```

# THE OPENWNS SOFTWARE DEVELOPER'S KIT (SDK)

## 3.1 Mastering your SDK (a guide to `playground.py`)

WNS consists of a number of sub-projects. Each of this project is configured in `config/projects.py`. The projects have different purposes. Some of them contribute as module to WNS, while others specify system tests. The system tests can be run to ensure the proper operation of WNS. All these projects are managed via Bazaar (see: <http://www.bazaar-vcs.org>). Since Bazaar has no built-in support to control multiple projects, `playground.py` has been developed to fulfill this task. Besides the management of the different project trees this script also performs WNS-wide tasks like compiling/installing all modules, building documentation and running tests as will be shown below.

`playground.py` offers a number of *commands* and according *modifiers*. A modifier modifies the way a command is executed (thus its name). All commands and modifiers are listed by executing:

```
$ ./playground.py --help
```

### 3.1.1 Compilation/Installation

One of the most important features of `playground.py` installs a version of WNS. You have probably used it right after check out. To install a version of WNS in `@c WNS/sandbox` issue the following command:

```
$ ./playground.py install --flavour=dbg
```

The string `dbg` tells `playground.py` to build the debugging version of WNS. It can be found in `WNS/sandbox/bin/dbg` and the modules are located under `WNS/sandbox/lib/dbg`. Installation flavours available are: - **dbg** (for debugging) - **opt** (optimized, for simulations, messages and asserts are disabled) - **profOpt** (like `opt` but with profiling support for `gprof`) - **callgrind** (to be used for CPU cycle profiling with `valgrind`)

There are also some modifiers available for the `install` command.

- **-flavour** Specify which build flavour to use (`dbg`, `opt`, ...)
- **-j/-jobs** Specify the number of compile jobs to be executed in parallel. Useful on multi-processor systems and when compiling distributed.
- **-static** All Modules will be linked into WNS
- **-scons** Forwards options (like `no-filter` or `no-inf`) to `scons`

- **-sandboxDir** Specify the location of your sandbox

### 3.1.2 Staying up-to-date

To upgrade the complete WNS including all sub-projects issue:

```
$ ./playground.py upgrade
```

For each (sub-)project `bzr pull` will be called. Any newly available patch in the repository will be applied. If you had changes on one of the projects, it can happen that conflicts occur. `@c playground.py` will stop and you should resolve the conflicts first.

To have an overview in advance where new patches are available for your WNS issue the following command:

```
$ ./playground.py missing
```

To see where conflicts can happen you need to find out which projects have local changes:

```
$ ./playground.py status
```

If you additionally specify `-diffs` with `status` the differences will be shown in detail for each project.

### 3.1.3 Cleaning up

From time to time it can be useful to clean up things (to remove old object files, etc). The following command assist you in that:

```
$ ./playground.py clean [objs, sandbox, docu, all]
```

The options have the following meaning:

- **objs**: remove the object files from the projects
- **sandbox**: remove the WNS installation (all libs, docu and binaries)
- **docu**: remove the directory `doxydoc` from each project
- **all**: all of the above

**Note:** All items which are removed here can be easily recreated. No hand-written code will be deleted.

**Note:** For the target `objs` you may need to additionally specify the modifier `-flavour` to remove the object file for a certain flavour (`dbg`, `opt`, ...)

### 3.1.4 Preparing Simulation Campaigns

Please see: `@ref simulation`

### 3.1.5 Creating Documentation

Freshening this docu after an upgrade.

```
$ ./playground.py docu
```

### 3.1.6 Running the test suite

To run the entire test suite do

```
$ ./playground.py runtests
```

This command runs all the tests (unit tests and system tests) for you. Before committing changes, you should *always* follow this step, to assure that your changes don't break anything.

While developing, you can also run tests individually. To run all unit tests enter:

```
$ cd tests/unit/unitTests/
$ ./wms-core -t
```

To run a certain unit test, use the option `-T` and give the name of the test you want to run, e.g.:

```
$ ./wms-core -t -T "rise::tests::BeamformingTest::simple"
```

If you want to configure certain parameters that are not explicitly configured in the test itself, you can add them either in the unittest config file (`tests/unit/unitTests/config.py`) or by executing

```
$ ./wms-core -t -T "rise::tests::BeamformingTest" \
-y "WNS.modules.rise.debug.antennas=True" \
-y "WNS.masterLogger.enabled=True"
```

To run a certain system test, go to the system test directory and execute

```
$ cd tests/system/WiMAC-Tests--main--1.0/
$ ./systemTest.py
```

In order to run only one specific test suite, execute the `wms-core` with the specific config file for the test suite. Note that the reference output will not be checked automatically!

```
$ cd tests/system/WiMAC-Tests--main--1.0
$ ./wms-core ./wms-core -f configSDMA.py
```

### 3.1.7 Customize the SDK contents (projects.py)

The openWNS SDK reads information on which modules to include from the file `config/projects.py`. This file is a plain Python file and can be edited. The structure of this file is

```
# Header
from wnsbase.playground.Project import *
import wnsbase.RCS as RCS

bzrBaseURL = "bzz://bazaar.comnets.rwth-aachen.de/openWNS/main"

# MODULE DEFINITIONS
```

```
# MODULE LIST
```

```
# PROCESSING INSTRUCTIONS
```

The module definition for the `openwns` executable looks like this:

```
wns_core      = Binary('./framework/wns-core--main--1.0',
                       "wns-core--main--1.0", bsrBaseUrl,
                       RCS.Bazaar('./framework/wns-core--main--1.0',
                                   'wns-core', 'main', '1.0'),
                       [libwns, ])
```

You create an instance of a project (see `wnsbase.playground.Project`). The most commonly used is `Library`, `Generic` or `Python`. Here we use `Binary` to tell the build-system to build an executable file. The first parameter is the path within the SDK where the project will be stored (`./framework/wns-core-main-1.0`). The second parameter tells playground where the remote branch is located. The string is append to `bsrBaseUrl` and is passed on to `bzr` if needed. You then pass in an instance of `RCS.Bazaar` where you need to repeat the path in the SDK and give three more parameters `modulename`, `series name` and `version`. Last but not least you need to tell `playground.py` if your project depends on another project within the SDK. Here `wns_core` depends on `libwns`. Note that `libwns` is an instance of `wnsbase.playground.Project.Library`.

After all module definitions the `projects.py` file contains a list `all` which includes all projects to be used. Make sure your project is included there. The `prereqCommands` is a list of `command directory` tuples. If you need some special setup commands you can place them here. `playground.py` executes the command in the specified directory during setup.

## 3.2 Creating and Maintaining Development Branches

**bring up to date for BZR!**

### 3.2.1 When work on a separate branch?

- You intend to develop a certain, cohesive feature in one of the WNS modules?
- It will take you a number of days to develop it?
- During this period you want to do interim commits whithout having to bother about leaving the mainline WNS in non-working state?
- You are tired of merging other people's changes with your changes before being able to commit them?

Then a private development branch may just be the right thing for you to use.

### 3.2.2 Bazaar and its Separation of Commit, Push and Pull Operation

Bazaar is a distributed version control system. The most important difference to CVS or to subversion is that it separates the procedure of making commits and the procedure of publication (called *push* in Bazaar) and retrieval (*pull*) patches to/from a remote location. A commit is always made locally to your copy of the branch (if you created a branch). If you want to publish your modifications you need to `@em` push it somewhere.

If you want to merge with your parent branch you need to push your changes back to the parent branch, i.e. the one you originally branched from. If you want to make it available to other users, but you do not want to merge simply

push your branches to another location. If there would be conflicts, then Bazaar will inform you and suggest that you do a @em merge.

### 3.2.3 Creating a branch from the WNS Mainline

By default the modules included in the openWNS SDK are already private branches. Each of the module is a full copy of its including all its history information. In fact, you do not need to create a private branch - you already have one.

### 3.2.4 Obtaining Mainline Patches

@todo to be written

### 3.2.5 Merging back

@todo to be written

## 3.3 Performance Tips

### 3.3.1 Distributed Compiling

The first compilation of openWNS after a fresh checkout can be very time consuming if it is performed on a single computer. Building a version for debugging and an optimized version can take from one to more than two hours depending on the capabilities of your computer.

If a network of computers is available (preferably all installed identically) and ICECC is available on these computers (see <http://en.opensuse.org/Icecream>) openWNS can be compiled distributed.

To enable distributed compilation the following options may be set in `config/private.py`:

```
privateEnv.Replace(ICECC='icecc')
privateEnv.configureCompiler(cc='/usr/bin/gcc', cxx='/usr/bin/g++')
```

**Note:** The statements must occur in this order. Otherwise icecc will not be used.

In order to visualize the distributed compiling, open the ICECC monitor on any computer that is part of the network, e.g., on 'myHost':

```
$ icemon &
```

### 3.3.2 Object Files

openWNS provides a way to put all object files into a separate directory. It is a good idea to put this directory onto another (fast) disk. When linking the libraries and executables the object files can be read from the other disk and are written to the disk where the SDK resides.

To enable this feature do the following:

```
$ mkdir pathToExternalObjDirOnOtherHardDisk
```

After this simply edit your `config/private.py` to contain the following line:

```
privateEnv.setExternalObjDir("pathToExternalExternalObjDirOnOtherHardDisk")
```

That's it. All object files will be written and read from there.

**Note:** This can also be used in case the left space on a partition is getting low.

# THE SIMULATION PLATFORM

## 4.1 The Event Scheduler

With the openWNS event scheduler you can schedule anything that is callable in your simulation. A callable can be a function pointer, a function object or anything else that implements the call operator, requires no arguments and has a return value of void. This page shows you how to use `boost::bind` to create such callables in a uniform way and use the resulting function objects with the scheduler.

### 4.1.1 Scheduling free functions

We will start with a very simple example. Suppose there exists a free function `freeFunction` within your simulation and you want to schedule calls to this function at later points in time. The example function we use here looks like this:

```
int globalVariable = 0;

void freeFunction()
{
    wns::logger::Logger log("tests", "SchedulerBestPractices");

    MESSAGE_BEGIN(NORMAL, log, m, "freeFunction is called.");
    m << "Setting globalVariable to 101";
    MESSAGE_END();

    globalVariable = 101;
}
```

Every time it is called it sets a global variable to the value 101. To schedule a call to this function at some later point in (simulation) time you could simply do:

```
wns::events::scheduler::Interface* scheduler = NULL;

scheduler = wns::simulator::getEventScheduler();

wns::events::scheduler::Callable c = &freeFunction;

scheduler->scheduleDelay(c, 10.0);
```

This tells the scheduler to call the function `freeFunction` after 10 seconds of simulation time have passed. The argument you pass to the `scheduleDelay` function is a `Callable` that carries a pointer to `freeFunction` and the delay after which the call to this function should be made. This is quite simple if you deal with free functions, i.e. functions that do not

belong to a class and therefore do not have a context. If you want to schedule calls to member functions (which you probably want to do frequently within an object oriented simulator) you need to be able to tell the scheduler on which object the scheduler should make the call to a member function, you want to bind the function pointer to an instance of an object. This is when you want to use `boost::bind`. Let us just stick with the last example for a short moment and see how you could use `boost::bind` to create a `Callable` for `freeFunction`.

```
wns::events::scheduler::Interface* scheduler = NULL;

scheduler = wns::simulator::getEventScheduler();

wns::events::scheduler::Callable c = boost::bind(&freeFunction);

scheduler->scheduleNow(c);
```

Pretty simple to do that. Simply call `boost::bind()` and pass the function pointer as the argument. This creates an object that is compatible with `wns::events::scheduler::Callable`. However, this does not have any immediate benefit for us. We will see later that by using `boost::bind` we get a consistent syntax for all `Callables` which greatly improves readability.

## 4.1.2 Scheduling Member Functions

So let us have a look at member functions now. Suppose you have the following class:

```
class ClassWithCallback
{
    int memberVariable_;

    wns::logger::Logger logger_;

public:
    ClassWithCallback() :
        memberVariable_(0),
        logger_("tests", "SchedulerBestPractices")
    {
    }

    void
    callback()
    {
        MESSAGE_BEGIN(NORMAL, logger_, m, "");
        m << "ClassWithCallback::callback is called."
          << "Setting globalVariable to 101";
        MESSAGE_END();
        this->memberVariable_ = 101;
    }

    int
    getMemberVariable()
    {
        return memberVariable_;
    }

    void
    setMemberVariable(int value)
    {
        memberVariable_ = value;
    }
};
```

```

    }
};

```

The behaviour of `ClassWithCallback::callback` is pretty much the same as the `freeFunction` of our last example, but now the scope is not global but limited to the scope of the class instance.

### 4.1.3 Scheduling Member Functions by Pointer

When we pass an instance of some type to a `boost::bind` expression it is copied. This is fine and yields the expected results if we work with pointers, but it does not necessarily yield the expected results if you pass your arguments by value (see next section). To schedule a member function of an object where you have a pointer to can be done like this:

```

wns::events::scheduler::Interface* scheduler = NULL;

scheduler = wns::simulator::getEventScheduler();

// Create a Smart Pointer to a new ClassWithCallback instance
ClassWithCallback* classWithCallbackPtr = new ClassWithCallback();

wns::events::scheduler::Callable c =
    boost::bind(&ClassWithCallback::callback,
               classWithCallbackPtr);

scheduler->scheduleDelay(c, 10.0);

```

### 4.1.4 Scheduling Member Functions by Value

The code below illustrates the usage of `boost::bind` if you have references to your callback objects. To avoid copying of your callback object when you pass it as an argument to `boost::bind`, just wrap in an `boost::ref` or `boost::cref` (wrapper for const references) object. Thereby, only the reference wrapper is copied but not the reference itself. You could also have simply taken the address (`&` operator) of your callback object and pass it to `boost::bind` to enforce pointer semantics.

```

wns::events::scheduler::Interface* scheduler = NULL;

scheduler = wns::simulator::getEventScheduler();

ClassWithCallback classWithCallbackInstance;

wns::events::scheduler::Callable c =
    boost::bind(&ClassWithCallback::callback,
               boost::ref(classWithCallbackInstance));

scheduler->scheduleDelay(c, 10.0);

```

### 4.1.5 Scheduling functions that take arguments

The examples above lack a very important feature. None of the above callbacks can take an argument. Most often within your simulation you want to schedule calls that take parameters. `Boost::bind` offers partial binding of parameters, too. If you bind all parameters in an `boost::bind` expression, the result is a nullary function object that you can pass to the scheduler. The following example illustrates this:

```
wns::events::scheduler::Interface* scheduler = NULL;

scheduler = wns::simulator::getEventScheduler();

ClassWithCallback* classWithCallbackPtr = new ClassWithCallback();

wns::events::scheduler::Callable c1 =
    boost::bind(
        &ClassWithCallback::setMemberVariable,
        classWithCallbackPtr,
        201);

wns::events::scheduler::Callable c2 =
    boost::bind(
        &ClassWithCallback::setMemberVariable,
        classWithCallbackPtr,
        302);

scheduler->scheduleDelay(c1, 11.0);

scheduler->scheduleDelay(c2, 10.0);
```

This concludes the scheduler best practices lesson.

### 4.1.6 Cancelling Events

Whenever you schedule an event by using either the `schedule` or `scheduleDelay` method of the `eventscheduler` these methods return an instance of `IEventPtr`. This is a handle for your scheduled event and can be used to remove it from the scheduler queue again. This is demonstrated in the next example.

```
wns::events::scheduler::Callable timeout = &freeFunction;

// Remember a handle of your event
IEventPtr timeoutHandle = scheduler->scheduleDelay(timeout, 15.0);

// Use the handle to cancel an event that was already scheduled
scheduler->cancelEvent(timeoutHandle);
```

## 4.2 Random Number Distributions

The openWNS random number distributions are based on the Boost Random library. openWNS provides a frontend to generate random numbers according to specific random distributions. There are many commonly used distributions available and can be found in the `wns::distribution` (`wns.Distribution` in Python) namespace. Available distributions are:

- Fixed (Deterministic)
- Negative Exponential
- Normal
- (Standard/Discrete) Uniform
- Binomial

- Geometric
- Erlang
- Poisson
- Pareto
- Rice

as well as arithmetic concatenations and truncated versions of them.

## 4.2.1 Constructing Random Number Distributions

Each distribution offers three constructors:

```
wns::distribution::Uniform::Uniform(double low,
                                     double high,
                                     wns::rng::RNGen* rng = wns::simulator::getRNG())

wns::distribution::Uniform::Uniform(const wns::pyconfig::View& config)

wns::distribution::Uniform::Uniform(wns::rng::RNGen* rng,
                                     const wns::pyconfig::View& config)
```

The first constructor takes the parameters of the distribution and a random number generator engine. Default engine is the global simulator random number engine. **You should not change this unless you really know what you are doing!** Taking a different engine could lead to unwanted correlations in your simulator run.

The second constructor is intended to be used with the `wns::distribution::DistributionCreator` static factory and is configured in Python. The corresponding Python classes can be found under `wns.Distribution`.

The third constructor allows to supply a custom random number generator engine using the `wns::distribution::RNGDistributionCreator`. **Above warnings apply.**

## 4.2.2 Drawing Random Numbers

Here is an example how to draw random numbers:

```
wns::distribution::StandardUniform stdUni;
std::cout << "Random Number from [0; 1.0] = " << stdUni() << "\n";
```

Random numbers are drawn using the `operator()`. They are always of type `double`. Note that `StandardUniform` has no parameters passed to the constructor. Its parameters are fixed.

## 4.2.3 Creating Distributions from the Static Factory

To make your code independent of a specific distribution you can use the static factory mechanism included in open-WNS. With this you can decide within your configuration which random number distribution is used during runtime without changing your C++ code.

You do not need to change anything in Python, so:

```
import wns.distribution
```

```
dis = wns.distribution.NegExp(4711.0)
conf.distribution = dis
```

In C++ you now need to use the plugin mechanism provided by the static factory. The code looks like this:

```
#include <WNS/distribution/Distribution.hpp>

wns::distribution::Distribution* dis;

// Of course you need to get the dist config variable
wns::pyconfig::View distConfig = config.get("dist");

// Use the plugin mechanism to create the distribution
std::string pluginName = distConfig.get<std::string>("__plugin__");
wns::distribution::DistributionCreator* dc =
    wns::distribution::DistributionFactory::creator(pluginName);
dis = dc->create(distConfig);
```

Use `RNGDistributionCreator` and `RNGDistributionFactory` to supply an own RNG to `dc->create`.  
**Above warnings apply**

### 4.2.4 Random Number Distribution Base Classes

Most distributions derive from `wns::distribution::IHasMean` Besides `operator()` to draw a random number they offer the `getMean()` method to return their mean value.

You might want to use something like this if you dynamically created a Distribution:

```
Distribution* dis = new StandardUniform();
double mean = dynamic_cast<wns::distribution::IHasMean*>(dis)->getMean();
```

Rice distribution does not offer this method and therefore does not derive from `wns::distribution::IHasMean`

### 4.2.5 Concatenation and Truncation of Random Number Distributions

Distributions can be concatenated or truncated using Python configuration:

```
import wns.Distribution

# Create triangle distribution by adding two uniform distributions
triangleDist = wns.Distribution.Uniform(0.0, 10.0) + wns.Distribution.Uniform(0.0, 10.0)

# Create a neg. exp. distribution shifted by 6.0 to the right
shiftedNegExp = wns.Distribution.NegExp(0.5) + 6.0

# Create a truncated normal distribution on (-inf; 15.0)
truncNorm = wns.Distribution.BELOW(wns.Distribution.Normal(0.0, 1.0), 15.0)
```

Available concatenations are `+` `-` `*` `/` resulting distribution offers the `getMean()` method.

Available truncation operators are `ABOVE` and `BELOW`. Resulting distributions **do not** offer the `getMean()` method.

## 4.2.6 Simulation Time Dependent Random Number Distributions

Distribution can depend on simulation time using `wns::distribution::TimeDependent`:

```
import wns.Distribution

dist = wns.Distribution.TimeDependent()
dist.eventList.append(wns.Distribution.Event(0.0, wns.Distribution.Uniform(40.0, 60.0)))
dist.eventList.append(wns.Distribution.Event(2.0, wns.Distribution.Normal(50.0, 20.0)))
dist.eventList.append(wns.Distribution.Event(5.0, wns.Distribution.Rice(0.0, 10.0)))
```

Depending on simulation time a different random number distribution is used. `getMean()` is not available.

## 4.2.7 Distributions Cheat Sheet

In Python you just need to instantiate a distribution from `openwns.distribution`, e.g.

You may select one of the following distributions:

Distribution	Configuration Usage Example
Fixed (Deterministic)	<code>openwns.distribution.Fixed(value = 2.34)</code>
Negative Exponential	<code>openwns.distribution.Fixed(mean = 100.112)</code>
Normal (Standard/Discrete)	<code>openwns.distribution.Normal(mean = 100.112, variance=10.6)</code> <code>openwns.distribution.Uniform(high = 11.0, low=9.0)</code>
Uniform	
Binomial	<code>openwns.distribution.Binomial(N = 10, p=0.3)</code>
Geometric	<code>openwns.distribution.Geometric(mean = 17.0)</code>
Erlang	<code>openwns.distribution.Erlang(rate = 1.23, shape=1.1)</code>
Poisson	<code>openwns.distribution.Poisson(mean=7.0)</code>
Pareto	<code>openwns.distribution.Pareto(shapeA = 0.4, scaleB = 0.3, xMin = 0.0, xMax = 1e100)</code>
Rice	<code>openwns.distribution.Rice(losFactor = 0.4, variance = 3.2)</code>

In C++ you can then create the distribution you just configured:

```
#include <WNS/distribution/Distribution.hpp>

wns::distribution::Distribution* dis;

// Of course you need to get the dist config variable
wns::pyconfig::View distConfig = config.get("dist");

// Use the plugin mechanism to create the distribution
std::string pluginName = distConfig.get<std::string>("__plugin__");
wns::distribution::DistributionCreator* dc =
    wns::distribution::DistributionFactory::creator(pluginName);
dis = dc->create(distConfig);

// Draw random numbers
double randomNumber = dis();
```

## 4.3 Measurement Sources

### 4.3.1 Introduction

openWNS decouples the generation and evaluation of measurements. The evaluation, may that be logging, filtering, sorting or simply storing them is described in detail in *Evaluation Framework*. In the following, the focus is solely on the generation of measurements and adding context information to them, which than may be processed by the evaluation framework.

### 4.3.2 Concept

In the following we will be making use of some fundamental concepts regarding the implementation of measurement sources, which will be defined now.

**Measurement Source** A measurement source is a logical concept that describes a location within the simulator where a measurement value is measured and is made available to the evaluation framework. A measurement value has a timestamp, which holds the time when the measurement value was generated. A measurement value may have context information.

**Measurement Value** A `double` value representing a measurement of some phenomenon in your simulator

**Context** A dictionary mapping an arbitrary but unique name to an `int` value. The context captures the state (within a certain scope) of your simulator when a measurement value is measured.

### 4.3.3 Realization

Measurement sources, values and their context information is realized by `wns::probe::bus::ProbeBus`. The `ProbeBus` realizes the connection of measurement sources to the evaluation framework. `wns::probe::bus::Context` realizes the context information.

Commonly, the entries in the context information of each measurement source do not change during the whole simulation (of course the value of each entry changes over time, but not the name). In such cases the `wns::probe::bus::ContextCollector` should be used to automatically generate the context information and the timestamp when a measurement source generates a measurement value and passes it on to the evaluation framework. The `wns::probe::bus::ContextProviderCollection` and `wns::probe::bus::ContextProvider` are used to define which context information is to be gathered by the `wns::probe::bus::ContextCollector`.

### 4.3.4 Implementing a Measurement Source

To add a measurement source to some class of your simulator you need to add

```
#include <WNS/probe/bus/ProbeBus.hpp>
#include <WNS/probe/bus/ProbeBusRegistry.hpp>
```

to the header file of that class. Suppose you have implemented a simple queueing model with a `Job` class that has some priority and that records the time when processing started. It may look like this:

```
class Job
{
public:
    enum Priority
```

```

{
    control = 0,
    realtime,
    multimedia,
    bestEffort
} priority;

Job();

Job(const Job&);

Priority priority_;

wns::simulator::Time startedAt_;

};

```

Furthermore, let us say that you have a class `Processor` that will process a `Job` and then generate a measurement of the processing time. The context of this measurement will be the `Job`'s priority class. The `Processor` class may look like this:

```

class Processor
{
public:

    Processor();

    void
    startJob(Job job);

    void
    onJobEnded(Job job);

private:
    boost::shared_ptr<wns::distribution::Distribution> dis_;

    wns::probe::bus::ProbeBus* probeBus_;
};

```

The class has two methods `startJob()` and `onJobEnded`. It has a distribution member variable to draw randomly distributed processing delays. To generate measurements we need a pointer to a `wns::probe::bus::ProbeBus`. This is your connection point to the evaluation framework. Let's have a look at the constructor of `Processor`.

```

Processor::Processor():
    dis_(new wns::distribution::Uniform(0.0, 1.0)),
    probeBus_(NULL)
{
    wns::probe::bus::ProbeBusRegistry* reg = wns::simulator::getProbeBusRegistry();
    probeBus_ = reg->getMeasurementSource("processor.processingDelay");

    assure(probeBus_ != NULL, "The probeBus_ is NULL");
}

```

To initialize our `ProbeBus` we ask the `wns::probe::bus::ProbeBusRegistry` to create one for us and register it with a unique name. Since we want to measure processing delay we choose the name `processor.processingDelay`. You should always prefix your measurement source name with the names-

pace of your class to make it unique. You will see in *Evaluation Framework* that this name is used to connect your evaluation to the measurement source.

Let's see what happens if someone request the processor to start a job. Here is `startJob`:

```
void
Processor::startJob(Job job)
{
    wns::events::scheduler::Interface* scheduler = wns::simulator::getEventScheduler();

    job.startedAt_ = scheduler->getTime();

    wns::simulator::Time processingTime = (*dis_)();

    wns::events::scheduler::Callable jobEndsCallable =
        boost::bind(
            &Processor::onJobEnded,
            this,
            job);

    scheduler->scheduleDelay(jobEndsCallable, processingTime);
}
```

We get the event scheduler and record the current time in the `job.startedAt_`. Then we draw a random processing time form the distribution and schedule a delayed call to our the `onJobEnded` method. If you do not understand how the scheduling works please read *The Event Scheduler* first.

So let's see how to generate the processing delay measurements. This is done int `onJobEnded`.

```
void
Processor::onJobEnded(Job job)
{
    // Create the context and populate it
    wns::probe::bus::Context context;

    context.insertInt("priority", job.priority_);

    // Get the current time
    wns::events::scheduler::Interface* scheduler = wns::simulator::getEventScheduler();

    wns::simulator::Time timestamp = scheduler->getTime();

    // Create the measurementValue
    double measurementValue = timestamp - job.startedAt_;

    assure(probeBus_ != NULL, "The probeBus_ is NULL");

    // Pass it all to the evaluation framework
    probeBus_->forwardMeasurement(timestamp,
                                   measurementValue,
                                   context);
}
```

Since we want to provide context information for our measurement value we first create an empty `wns::probe::bus::Context` and populate it with the job's priority which we read from the `Job` itself. Then we get the current time from the scheduler and calculate the measurement value, which is simply the current time minus the job's start time, i.e. the processing delay.

Everything is prepared now. We have the *measurement value*, the *timestamp* and the *context*. To pass it on to the evaluation framework, we first assure that `probeBus_` is not NULL and then call `forwardMeasurement` of our `probeBus_`.

## 4.4 Context Collector

Filling the context of a measurement by hand can be cumbersome. Especially, when you want to collect context from different locations in your simulator. This is where the `ContextCollector` can help you. We will follow the example of the previous section and extend it to use the `ContextCollector`. You will see that the code for filling the context can be organized in a way that makes the particular structure of a context reusable.

### 4.4.1 Concepts

**Context Provider** A Context Provider can provide context for measurement when asked.

**Context Provider Collection** The name says it all. A collection of Context Providers.

**Context Collector** Collects context. Additionally it wraps a `ProbeBus` and takes care of fetching the `context` and `timestamp` for the call to `forwardMeasurement`. You only need to provide the measurement value.

### 4.4.2 Simple use of the Context Collector

First of all we need to include the `ContextCollector`'s header file.

```
#include <WNS/probe/bus/ProbeBus.hpp>
#include <WNS/probe/bus/ProbeBusRegistry.hpp>
#include <WNS/probe/bus/ContextCollector.hpp>
```

Ok. This was not so hard. We will leave the `Job` class unchanged, but derive it from `wns::RefCountable`. This is needed because we will use `wns::SmartPointer` to do the memory management for all `Job` instances.

```
class Job:
    virtual public wns::RefCountable
{
public:
    enum Priority
    {
        control = 0,
        realtime,
        multimedia,
        bestEffort
    } priority;

    Job();

    Job(const Job&);

    Priority priority_;

    wns::simulator::Time startedAt_;
};
```

We add a `ContextCollector` as private member to the `Processor` class. The `ContextCollector` simplifies collecting measurements. It takes care of filling the context and also take care to pass the current simulation time to the underlying `ProbeBus`.

```
class Processor
{
public:

    Processor();

    void
    startJob(const wns::SmartPtr<Job>& job);

    void
    onJobEnded(const wns::SmartPtr<Job>& job);

    int
    getID();

private:
    int
    generateID();

    boost::shared_ptr<wns::distribution::Distribution> dis_;

    wns::probe::bus::ContextCollector processingDelayCC_;

    int id_;
};
```

The constructor of a the `ContextCollector` takes the name of the measurement source as a parameter. There is a more complex constructor that is described in the second part of this chapter.

```
Processor::Processor():
    dis_(new wns::distribution::Uniform(0.0, 1.0)),
    id_(generateID()),
    processingDelayCC_("processor.processingDelay")
{
}
```

If you want to publish a measurement you simply use the `put` method of the `ContextCollector`. The first argument is the measurement value. If you want to provide some additional context you can pass a `boost::tuple` (see `boosttuple`) as second parameter. Here we use `boost::make_tuple` to construct a tuple. The entries with odd indices (starting at 1) of the tuple are the keys and must always be of type `std::string` the even ones are the values and can either be of type `int` or `std::string`.

```
void
Processor::onJobEnded(const wns::SmartPtr<Job>& job)
{
    // Create the measurementValue
    double measurementValue =
        wns::simulator::getEventScheduler()->getTime() - job->startedAt_;

    // Pass it all to the evaluation framework
    processingDelayCC_.put(measurementValue,
        boost::make_tuple("priority", job->priority_,
```

```

        "processorID", this->id_));
}

```

If you compare this to the `Processor::onJobEnded` method of the last example, you can see that you basically need two lines of code where you previously needed around 10.

### 4.4.3 Using and Implementing Context Providers

If you want to add context from other locations in your simulator without introducing tight coupling between the entity that provides the context and the entity that provides the measurement you should make use of `ContextProviders`. A `ContextProvider` encapsulates the provisioning of context entries in a separate class. This section shows you how to implement a `ContextProvider` and add it to a `ContextProviderCollection` which defines the `Context` of a `ContextCollector`. So lets get started. Again, we first need to include the `ContextCollector`'s header file.

```

#include <WNS/probe/bus/ProbeBus.hpp>
#include <WNS/probe/bus/ProbeBusRegistry.hpp>
#include <WNS/probe/bus/ContextCollector.hpp>

```

Ok. This was not so hard. We will leave the `Job` class unchanged, but derive it from `wns::osi::PDU` instead of `wns::RefCountable?`. `openWNS` is a system level simulation tool that focusses on protocol behaviour. The `ContextCollector` expects jobs to be PDUs. So it this is a reasonable approach.

```

class Job:
    public wns::osi::PDU
{
public:
    enum Priority
    {
        control = 0,
        realtime,
        multimedia,
        bestEffort
    } priority;

    Job();

    Job(const Job&);

    Priority priority_;

    wns::simulator::Time startedAt_;
};

```

We will make some extensions to the `Processor` class. First of all the signatures of `startJob()` and `onJobEnded()` now take a `wns::SmartPtr<Job>` as argument. This makes it more realistic, because normally a `Job` traverses many different classes in your simulator. None of these classes knows when to delete the `Job`, therefore memory management is handled by a `SmartPtr`.

```

class Processor
{
public:

    Processor();

```

```
void
startJob(const wns::SmartPtr<Job>& job);

void
onJobEnded(const wns::SmartPtr<Job>& job);

int
getID();

private:
int
generateID();

wns::probe::bus::ContextProviderCollection
getContextProviderCollection();

boost::shared_ptr<wns::distribution::Distribution> dis_;

wns::probe::bus::ContextCollectorPtr processingDelayCC_;

int id_;
};
```

You can see that there is a public method `getID()` and a private method `generateID()` as well as a private member `id_`. We assume that there will be many processor in a simulation run which will be true for any non-trivial simulation. Each processor has a unique `processorID` which you can get by using `getID()`. During construction of a processor a new unique `processorID` is automatically generated by `generateID()`. Here is the code for it:

```
int
Processor::generateID()
{
    static int lastID = 0;

    lastID++;

    return lastID;
}
```

What we will do now is to create a context provider collection during startup of each processor. Then a context collector will be initialized to use this collection. The collection will contain two context providers. One that can read the `priority` from the `Job` that is evaluated, the other will callback the processor to provide its own `processorID`. Both context entries will then be passed to the evaluation framework.

Let's start by implementing the `PriorityProvider` that reads the priority of a job. Here is the class declaration:

```
class PriorityProvider:
    public wns::probe::bus::PDUContextProvider<Job>
{
public:
    PriorityProvider(std::string key);

private:
    virtual void
    doVisit(wns::probe::bus::IContext& c, const wns::SmartPtr<Job>& job) const;

    std::string key_;
};
```

We derive `PriorityProvider` from a template class `wns::probe::bus::PDUContextProvider`. The template parameter is the `Job` class. This will take care of some basic conversion for us. On construction we pass a key to the provider. This will be used when making entries in a context. The `PDUContextProvider` is an abstract class that forces us to implement the `doVisit` method. This method is called whenever a `ContextCollector` needs to construct context for a measurement value. It is rather simple:

```
void
PriorityProvider::doVisit(wns::probe::bus::IContext& c,
                        const wns::SmartPtr<Job>& job) const
{
    c.insertInt(key_, job->priority_);
}
```

The Visitor Pattern is used here. The context is passed to each context provider as reference, so it can be written by each of them. The current job is also passed along, but unlike the context it is passed in read-only mode. The only thing we do here is to insert the job's priority into the context and give it the name `key_`, which is set upon construction.

To provide the `processorID` we will use a generic purpose provider that can be used whenever a simple property of some object is to be read. So basically now we have the context providers ready and can start defining the context collector and its context provider collection. Let's have a look at the constructor of `Processor`.

```
Processor::Processor():
    dis_(new wns::distribution::Uniform(0.0, 1.0)),
    id_(generateID())
{
    processingDelayCC_ = wns::probe::bus::ContextCollectorPtr(
        new wns::probe::bus::ContextCollector(
            this->getContextProviderCollection(),
            "processor.processingDelay"));
}
```

Here, the `processingDelayCC_` is constructed. This is the context collector for our processing delay. Focus on the inner construction of the `ContextCollector`. To ease the memory management we wrap that object by a smart pointer `ContextCollectorPtr`. There is no magic about that.

The constructor of a *ContextCollector* takes a context provider collection as its first argument, and the name of its measurement source a second argument. The context provider collection is constructed by `getContextProviderCollection()`. Which is defined as:

```
wns::probe::bus::ContextProviderCollection
Processor::getContextProviderCollection()
{
    wns::probe::bus::ContextProviderCollection cpc;

    PriorityProvider jobPriorityProvider("priority");

    wns::probe::bus::contextprovider::Callback
        processorIDProvider("processorID",
                           boost::bind(&Processor::getID, this));

    cpc.addProvider(jobPriorityProvider);
    cpc.addProvider(processorIDProvider);

    return cpc;
}
```

First of all we construct an empty `ContextProviderCollection` named `cpc`. We construct one of our `PriorityProvider`'s and then use the general purpose `Callback` context provider. This provider takes the name of the measurement source as first argument. The second argument is any callable that returns an integer and takes zero arguments. We use `boost::bind` to tell `Callback` that whenever it needs context information it should call the `getID` method of `this` instance. Then we add both providers to the context provider collection and return it to the caller. This finishes the setup of the context provider collection and the context collector. How can it be used? Here is the new code of `onJobEnded()`.

```
void
Processor::onJobEnded(const wns::SmartPtr<Job>& job)
{
    // Create the measurementValue
    double measurementValue =
        wns::simulator::getEventScheduler()->getTime() - job->startedAt_;

    // Pass it all to the evaluation framework
    processingDelayCC->put(job, measurementValue);
}
```

As you can see it became much shorter. We only calculate the processing time and the call `put`. The first argument is the job and the second the actual measurement. The `ContextCollector` takes care that the job is passed to the `PriorityProvider` and that the current `processorID` is retrieved. We have successfully captured all the code to create context in the constructor of `Processor`. The context provider collection can be copied, passed around and reused and even hierarchically organized. This will be demonstrated in the next sections.

**Note:** todo: Describe the usage of the `ContextCollector` within the `Node/Component` concept.

## 4.5 Evaluation Framework

Describe the following:

- Introduction
- Concepts \* `TreeNode` \* `TreeNodeSet` \* Generators
- Quickstart
- HowTos #. Construct a new evaluation tree #. Working with `TreeNodeSets` #. Write your own Generators #. Advanced : Tagging `NodeSets`
- Internals #. Replacement of `ProbeBusRegistry`

### 4.5.1 Introduction

The openWNS Evaluation Framework built on top of the `ProbeBus` implementation. A short recap of the `ProbeBus` concept is given here to help you understand the basic concepts. For an in-depth understanding of the evaluation framework you should also make yourself familiar with the `ProbeBus` framework.

- `ProbeBus` : A `ProbeBus` may be observed by other `ProbeBusses`. A `ProbeBus` forwards the current simulation time, a measurement and context information to its observers. A `ProbeBus` may accept only a subset of the forwarded information depending on the provided context. In this way tree structures can be realized to sort measurements.

- Context : The context of a measurement is additional information that is provided to allow for sorting of measurements, e.g. when measuring SINR values it the context could be the current position of the terminal. In this way the spatial SINR distribution could be evaluated.
- MeasurementSource : A ProbeBus within your simulator where a measurement originates from and is published to subsequent ProbeBusses. A MeasurementSource has a unique name for identification.

## Concepts

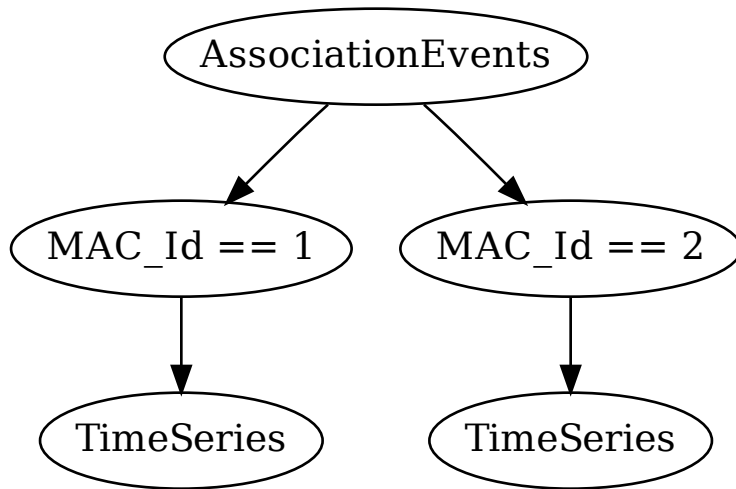
The evaluation framework of openWNS is used to build trees of ProbeBusses. The whole evaluation framework is only used during configuration time (in Python). The tree structures are mapped to trees of ProbeBusses at run-time. You will encounter three concepts when working with the evaluation framework.

- `pycoshort{openwns.evaluation.TreeNode}` : The fundamental concept. Represents a node within a tree. A `TreeNode` may have a parent and it may have multiple children.
- `pycoshort{openwns.evaluation.TreeNodeSet}` : Contains several `TreeNodes`. One `TreeNode` is only included once.
- `pycoshort{openwns.evaluation.generators}` : A generator creates new tree nodes.

### 4.5.2 Quickstart

The openWNS Evaluation Framework is used to define how measurements are sorted according to their Context. For example, say you have a measurement source 'AssociationEvents' in your simulation model and each station (mobile and base station) provides the MAC.Id context with its own unique MAC-ID. Assume that you want to have a time series for every base station of its association events. Here is how to define this in your configuration by using the openwns evaluation framework.

```
bsIdList = [1, 2]
node = openwns.evaluation.createSourceNode(sim, 'AssociationEvents')
node.appendChildren(Separate(by = 'MAC.Id', forAll = bsIdList, format='MAC_ID%d'))
node.getLeafs().appendChildren(TimeSeries())
```



After your simulation has finished you will find two files in your output directory, which contain the respective time series. The files are named:

```

AssociationEvents_MACId1_Log.log.dat
AssociationEvents_MACId2_Log.log.dat
  
```

Every file starts with its measurement source name. The Separate generator appends for each ID a suffix according to the format specifier. In this case MACId1 and MACId2 are appended. For historical reasons the TimeSeries generator appends the file suffix 'Log.log.dat'. Suffixes are separated by an underscore character.

### 4.5.3 HowTos

#### Creating a new evaluation tree

To create a new evaluation tree for a measurement source you should first get the appropriate source node.

```
node = openwns.evaluation.createSourceNode(sim, 'SINR')
```

This will create a new TreeNode and attaches it to the measurement source 'SINR'. If a TreeNode for this measurement source already exists, it will instead return that TreeNode (possibly including all the child nodes already configured).

#### Basic Evaluation

Now lets attach some evaluation to this measurement source.

```

from openwns.evaluation import
node = openwns.evaluation.createSourceNode(sim, 'SINR')
node.addChildren(Moments())
  
```

This will attach an evaluation node to this measurement source that reports basic statistics of ALL measurements. The Moments generator creates exactly one TreeNode which evaluates the basic statistics.

```
SINR_Log.dat
```

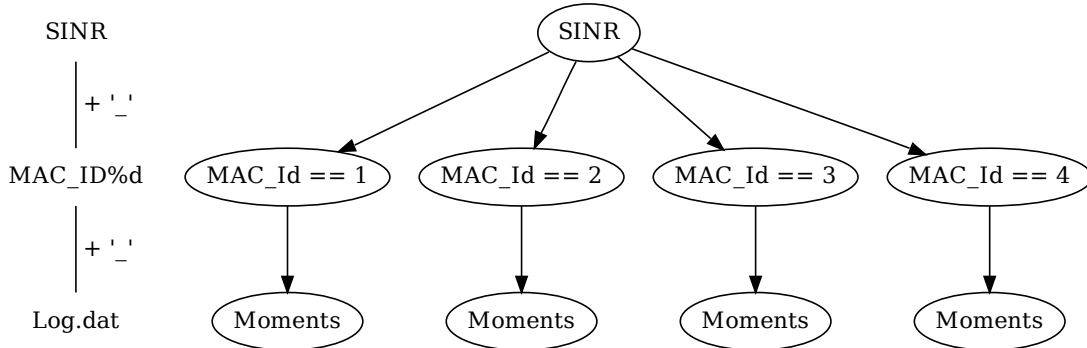
### Basic Sorting

Now lets do some sorting. Suppose all your SINR measurements have a context of your stations MAC\_ID. We now collect the basic statics for each of your station. In this example we will have 4 Stations with MAC\_IDs 1,2,3 and 4.

```
from openwns.evaluation import
node = openwns.evaluation.createSourceNode(sim, 'SINR')
node.addChildren(Separate(by = 'MAC_ID', forAll = [1,2,3,4], format='MAC_ID%d'))
node.getLeafs().addChildren(Moments())
```

The Separate generator creates one TreeNode for each entry in the list 'forAll'. Each of these nodes only accepts measurements where the context entry 'MAC\_ID' is of the respective value. The getLeafs() method returns a NodeTreeSet of all leafs of the tree. Calling addChildren() on a NodeTreeSet will use the generator to create new nodes for each TreeNode in the NodeTreeSet. Here, 4 TreeNodes that evaluate the basic statistics will be created in total. The resulting tree looks like this.

Filename:



You also have some control to the naming of your files. Filenames are constructed by concatenating the format strings of each TreeNode starting from the root node and then going down to the child nodes. For example, the leftmost Moments TreeNode will have a filename which is the concatenation of the format string of the root TreeNode ('SINR', you do not have control on that) and the string 'MAC\_ID1' and a string 'Log.dat', which is attached by the Moments TreeNode itself. Each of these parts of the filename are concatenated by an underscore character.

After your simulation has finished you will have four files in your output directory which contain the SINR statistics for each station (i.e. for each MAC\_ID).

```
SINR_MAC_ID1_Log.dat
SINR_MAC_ID2_Log.dat
SINR_MAC_ID3_Log.dat
SINR_MAC_ID4_Log.dat
```

## Memory and CPU time probing

The simulation library provides probing output of memory consumption and ratio of simulation time to real time. Following must be added to the configuration to write the output of those probes:

```
import openwns.evaluation.default
```

```
openwns.evaluation.default.installEvaluation(openwns.simulator.getSimulator())
```

This will create four files in the output directory:

- wns.Memory\_Moments.dat
- wns.Memory\_TimeSeries.dat
- wns.SimTimePerRealTime\_Moments.dat
- wns.SimTimePerRealTime\_TimeSeries.dat

Moments probes contain the statistics over the whole simulation run, TimeSeries probes show the current memory consumption and current simulation time divided by total runtime of the simulator. They are updated using the status writer. Update frequency can be adjusted by the following line in the configuration:

```
openwns.simulator.getSimulator().statusWriteInterval = 30 # in seconds realTime
```

The interval should not be chosen too low or else the simulator will be busy writing output to disk all the time.

## 4.5.4 Writing your own Generators

```
from openwns.evaluation.generators import
from openwns.evaluation.tree import
from openwns.evaluation.wrappers import
```

```
import openwns.probebus
```

```
class SeparateByQoSClass(ITreeNodeGenerator):
```

```
    def __init__(self, format="QoS%s"):
        self.format = format
        self.contextKey = "QoSClass"
        self.QoSClasses = []
        self.QoSClasses[0] = 'background'
        self.QoSClasses[1] = 'streaming'
        self.QoSClasses[2] = 'conversational'
        self.QoSClasses[3] = 'control'
```

```
    def __call__(self, pathname):
        for i in len(self.QoSClasses):
            probebus = openwns.probebus.ContextFilterProbeBus(self.ContextKey,
                                                                [ self.QoSClasses[i] ]
                                                                )
```

```
            wrapper = wrappers.ProbeBusWrapper(probebus,
                                                self.format % self.QoSClasses[i])
```

```
            treeNode = tree.TreeNode(wrapper)
```

```
yield treeNode
```

## 4.6 Smart Pointers

### 4.6.1 Motivation

As classes get created dynamically memory management, especially releasing memory occupied by data structured not used anymore becomes an issue. The C++ programming language uses the keywords `new` and `delete` for this. In most cases this task is simple. A new object is created within some existing class instance and stored there. Many times this is done at startup. So the class instance, which created the new object, will also delete it, most likely at the end of its own lifetime.

Things get more complicated if objects are created and “passed” around to other objects. An example for that are protocol data units (PDUs), in the following called packets, in a network simulator. Some source object in a traffic generator creates packets and then passes them down the protocol stack. At the receiving site packets are passed up the stack until they reach a sink. Now who should delete a packet once it is not needed anymore? The source? But how can it know if the packet was received. The sink? What if the packet gets lost during transmission and never reaches the sink? What if it is a broadcast or multicast packet received by multiple sinks?

There could be many more possible places that could be potentially used to delete the packet object, but they all have drawbacks. The answer on when the packet object should be deleted: “When nobody needs it anymore”.

To achieve this, the packet object itself needs to keep track of how many other objects are referencing it. Once it discovers that there is nobody referencing it, it can safely call the delete method.

### 4.6.2 Usage Example

In the queuing network example `WNS/queuing/MM1StepX.cpp` job objects of class `wns::queuing::Job` are created and stored in a Smart Pointer of type `wns::queuing::JobPtr`. The Smart Pointer declaration is just a typedef to `wns::SmartPtr<Job>`. Jobs are created using the `new` in `onCreateJob` and stored in the Smart Pointer using its constructor. They are then stored in the queue. After a job is processed in `onJobProcessed` there is no need to call the delete method. After the job is extracted from the queue the queue does not reference it anymore. It is only referenced by the local variable `XXX` in the `onJobProcessed` method. Once this methods returns the lifetime of `XXX` ends. Now there is no reference to the job anymore. The Smart Pointer will then automatically release the memory occupied by the job.

### 4.6.3 Accessing Smart Pointers

Smart Pointers must be accessed like any other pointer. Members are accessed using the “->” operator. The object referenced by the Smart Pointer is accessed using the star “\*” operator.

### 4.6.4 Creating Own Smart Pointers

A class `MyClass` which should be stored in a Smart Pointer needs to derive from `wns::RefCountable`. This assures it is able to keep track how often it is referenced. The Smart Pointer itself is declared as `wns::SmartPtr<MyClass>`. It is common to provide a typedef in `MyClass.hpp`. `typedef wns::SmartPtr<MyClass> MyClassPtr;`

## 4.6.5 Notes on Smart Pointers

Objects kept in Smart Pointers should never be also accessed using normal pointers. The Smart Pointer does not have knowledge about normal pointers referencing the object and could delete it while it is still needed by code using the normal pointers.

Sorting Smart Pointers: When storing Smart Pointers in a sorted container you most likely want to use the comparison method of the class the Smart Pointer points to. You therefore need to provide XXX as the sorting method which will automatically call the intended sorting method.

Circular references: A Smart Pointer should never keep another Smart Pointer to itself or to another class pointing back to itself. Example: Imagine a class for a double linked list keeping a pointer to the next and previous object in the list. Object 1 has a next pointer to object 2 and object 2 a previous pointer to object 1. If object 1 gets removed from the list occupied memory will not be released since it is still referenced by object 2. The memory occupied by object 2 will never be released since object 2 is referenced by object 1.

## 4.6.6 Debugging Smart Pointers

ToDo

## 4.7 How to write a Finite State Machine

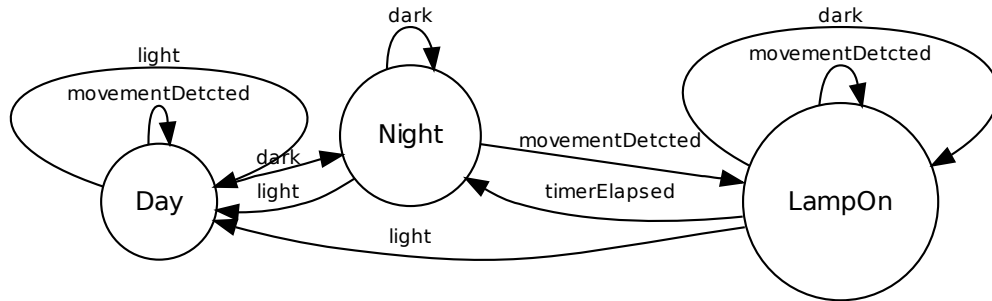
A simple example should clarify how to implement a finite state machine based on the `wns::fsm::FSM` template. A finite state machine for a light control, which switches the light at night on if a movement was detected and switches it off again after a certain timer has elapsed should be designed. The following sections will show the necessary steps to implement the light control based on `wns::fsm::FSM`. The following sub-sections represent the necessary steps:

### 4.7.1 Modelling the FSM Light Control

**Recipe:** Find a model which describes the problem based on a finite state machine

First of all we need to find a model for the finite state machine. We know that the Lamp Control should turn on the lamp if a movement was detected and if it's currently night. Thinking about this we can derive that we need at least three states for the FSM to represent the different behaviours: Day, Night, LampOn, since the FSM must react different to input if it's day or night and different again, if the light is currently turned on (e.g., at day the light must not be turned on if a movement was detected).

At run time the finite state machine will be in one of these three states. Being in these states it needs to react to certain signals (input) with some action. The action may also include a state change (e.g., the finite state machine is in state "Night" and the signal "light" arrives, the state would change to "Day"). The signals needed here are: dark (to switch from Day to Night), light (to switch from Night to Day), movementDetcted (to switch the Lamp on) and timerElapsed (to switch the lamp off after some time). With these states and signals, the model of our finite state machine "Lamp Control" looks like this:



Summary of the model:

- It has three states: #. Day #. Night #. LampOn
- It has four signals: #. light #. dark #. movementDetcted #. timerElapsed

After this model has been identified it needs to be transferred to C++. The framework provided by `wns::fsm::FSM` will help us to get the model implemented fast and efficiently. Most of the following steps contain a sentence describing in recipe style what needs to be done.

#### 4.7.2 An interface (abstract class) to define the signals

**Recipe:** Write a class having all signals of the finite state machine as pure virtual functions.

The signals have to be modeled as pure virtual methods of a class. Make sure the class is really an interface (does not contain any variables, is stateless). The States classes will be derived from this interface later. The interface for our signals looks like this:

```

class LightControlSignals
{
public:
    // the four signals as pure virtual functions
    virtual LightControlSignals*
    light() = 0;

    virtual LightControlSignals*
    dark() = 0;

    virtual LightControlSignals*
    movementDetected() = 0;

    virtual LightControlSignals*
    timeElapsed() = 0;

    // abstract interface with virtual functions needs
    // virtual destructor (otherwise g++ complains)
    virtual
    ~LightControlSignals()
    {}
}
  
```

```
protected:
    // only called from derived classes
    LightControlSignals()
    {}

private:
    LightControlSignals(const LightControlSignals&);
};
```

**Note:** The copy constructor has been disallowed in order to prevent copying of states. The default constructor is protected: since an interface can't be instantiated the constructor can only be called by a derived class.

### 4.7.3 States share Variables

**Recipe:** Write a class containing all the variables of the FSM shared by the different states

The finite state machine is also characterized by a set of variables which it holds. All these variables have to be implemented in a struct or class which will be used by the `wns::fsm::FSM` template later. For simplicity our example assumes the class defining the variables is called "Lamp" and contains only one variable called "on":

```
struct Lamp
{
    Lamp() :
        on(false)
    {}

    bool on;
};
```

### 4.7.4 An interface for the FSM "Light Control"

**Recipe:** Make a handy typedef to the interface of your FSM

The template `wns::fsm::FSM` instantiated with `Variables` and `Signal` will not implement the finite state machine itself. It will provide an interface for the final finite state machine. To have a handy alias create a typedef to this interface with the signal interface from above ("LightControlSignals") and the variables that characterize the finite state machine ("Lamp"):

```
typedef FSM<LightControlSignals, Lamp> LightControlInterface;
```

### 4.7.5 Implementation of the finite state machine

**Recipe:** Derive the final implementation from the interface (aliased by the typedef)

We're going to use the previously introduced typedef defining our FSM to derive a real implementation of this FSM. All methods of the signal interface ("LightControlSignals") need to be defined here:

```
class LightControl :
    public LightControlInterface
{
public:
    LightControl(const LightControlInterface::VariablesType& v);
```

```

    void
    movementDetected();

    void
    timeElapsed();

    void
    light();

    void
    dark();

    Lamp&
    lamp()
    {
        return getVariables();
    }
};

```

The implementation of the methods forward the method call simply to the current state object:

```

FSMTest::LightControl::LightControl(const FSMTest::LightControlInterface::VariablesType& v) :
    FSMTest::LightControlInterface(v)
{
    changeState(createState<Night>());
}

void
FSMTest::LightControl::movementDetected()
{
    changeState(getState()->movementDetected());
}

void
FSMTest::LightControl::timeElapsed()
{
    changeState(getState()->timeElapsed());
}

void
FSMTest::LightControl::light()
{
    changeState(getState()->light());
}

void
FSMTest::LightControl::dark()
{
    changeState(getState()->dark());
}

```

**Note:** The constructor must always take a const reference to the type exported as “VariablesType” by the FSM interface (“LightControlInterface”, the typedef)

**Note:** The constructor must set the FSM to a valid initial state (here Night).

## 4.7.6 Implementing the behaviour in the different states

**Recipe:** Implement a class for each state

Each state is represented by a different class. These classes will be instantiated and deleted by the FSM. Here the definition of the states (Day, Night and LampOn) are presented:

```
class Day :
    public LightControlInterface::StateInterface
{
public:
    Day(LightControlInterface* lci) :
        LightControlInterface::StateInterface(lci, "wns_fsm_tests_Day")
    {}

    virtual StateInterface*
    movementDetected();

    virtual StateInterface*
    timeElapsed();

    virtual StateInterface*
    light();

    virtual StateInterface*
    dark();
};

class Night :
    public LightControlInterface::StateInterface
{
public:
    Night(LightControlInterface* lci) :
        LightControlInterface::StateInterface(lci, "wns_fsm_tests_Night")
    {}

    virtual StateInterface*
    movementDetected();

    virtual StateInterface*
    timeElapsed();

    virtual StateInterface*
    light();

    virtual StateInterface*
    dark();
};

class LampOn :
    public LightControlInterface::StateInterface
{
public:
    LampOn(LightControlInterface* lci) :
        LightControlInterface::StateInterface(lci, "wns_fsm_tests_LampOn")
    {}
};
```

```

    virtual StateInterface*
    movementDetected();

    virtual StateInterface*
    timeElapsed();

    virtual StateInterface*
    light();

    virtual StateInterface*
    dark();
};

```

**Note:** All methods from the signal interface must be redefined.

**Note:** The state must be derived from an type exported by the interface of the FSM (“Light Control Interface”, the typedef) named “StateInterface”

**Note:** The constructor must take a pointer to the interface of the FSM

The implementation of the methods of these classes looks like this:

```

STATIC_FACTORY_REGISTER_WITH_CREATOR(FSMTest::Day,
                                     FSMTest::LightControlInterface::StateInterface,
                                     "wns_fsm_tests_Day",
                                     FSMConfigCreator);

```

```

FSMTest::Day::StateInterface*
FSMTest::Day::movementDetected()
{
    // ignore
    return this;
}

FSMTest::Day::StateInterface*
FSMTest::Day::timeElapsed()
{
    throw(Exception("Signal only valid in state LampOn"));
    return this;
}

FSMTest::Day::StateInterface*
FSMTest::Day::light()
{
    // ignore
    return this;
}

FSMTest::Day::StateInterface*
FSMTest::Day::dark()
{
    return getFSM()->createState<Night>();
}

```

```

STATIC_FACTORY_REGISTER_WITH_CREATOR(FSMTest::Night,
                                     FSMTest::LightControlInterface::StateInterface,
                                     "wns_fsm_tests_Night",
                                     FSMConfigCreator);

```

```

FSMTest::Night::StateInterface*
FSMTest::Night::movementDetected()
{
    vars().on = true;
    return getFSM()->createState<LampOn>();
}

FSMTest::Night::StateInterface*
FSMTest::Night::timeElapsed()
{
    throw(Exception("Signal only valid in state LampOn"));
    return this;
}

FSMTest::Night::StateInterface*
FSMTest::Night::light()
{
    return getFSM()->createState<Day>();
}

FSMTest::Night::StateInterface*
FSMTest::Night::dark()
{
    // ignore
    return this;
}

```

```

STATIC_FACTORY_REGISTER_WITH_CREATOR(FSMTest::LampOn,
                                     FSMTest::LightControlInterface::StateInterface,
                                     "wns_fsm_tests_LampOn",
                                     FSMConfigCreator);

```

```

FSMTest::LampOn::StateInterface*
FSMTest::LampOn::movementDetected()
{
    // ignore
    return this;
}

FSMTest::LampOn::StateInterface*
FSMTest::LampOn::timeElapsed()
{
    vars().on = false;
    return getFSM()->createState<Night>();
}

FSMTest::LampOn::StateInterface*
FSMTest::LampOn::light()
{
    vars().on = false;
    return getFSM()->createState<Day>();
}

FSMTest::LampOn::StateInterface*
FSMTest::LampOn::dark()
{
    // ignore

```

```
        return this;  
    }
```

**Note:** The name of a state is defined by the string used to register the state at the static factory and has to be unique throughout the WNS. This name has to be used also in the constructor of the state.



# WRITING CODE

## Todo

This is another todo entry

## 5.1 Writing Documentation

Documentation is written either in Doxygen style, inline with the source code, for the API documentation or in reStructured Text for this handbook.

### 5.1.1 Extracting Examples

Whenever you write documentation for a piece of code it should remain valid throughout the lifetime of that code. If the code changes, the documentation should be adopted. However, this is a tedious process and cannot be automated. Human beings often forget to update the documentation whenever their code changes.

Therefore, it is explicitly encouraged to extract examples from unit tests that are contained within the main test suite. The commit policy enforces that all tests in the main test suite must succeed before committing. If you extract your examples from these tests, then it is quite clear that all your examples will either remain valid or will be changed by developers if they break them.

You can mark a code snippet by using the `begin example` and `endexample` environment. The examples will be automatically extracted by `playground.py` and will be made available to Doxygen and Sphinx for reference. Below is an example of this.

```
// begin example "wns.avaragetest.header.example"
#include <WNS/Average.hpp>
#include <WNS/TestFixture.hpp>

namespace wns { namespace tests {
    class AverageTest : public CppUnit::TestFixture {
        CPPUNIT_TEST_SUITE( AverageTest );
        CPPUNIT_TEST( testPutAndGet );
        CPPUNIT_TEST( testReset );
        CPPUNIT_TEST_SUITE_END();

    public:
        void setUp();
        void tearDown();
        void testPutAndGet();
        void testReset();

    private:
        Average<double> average;
    };
};
```

```
        };  
    }  
    // end example
```

## 5.1.2 Including Examples in the API Documentaion

You can include an example in some other doxygen documentation by using the `@include` statement of Doxygen. Just reference your example by its name and it will be rendered in place.

```
/**  
 * @page wns_AboutTheAverageTest About the Average Test  
 *  
 * The @c AverageTest is used to test @c wns::Average. Its class  
 * definition is as follows:  
 *  
 * @include wns.avaragetest.header.example  
 */
```

## 5.1.3 Including Examples in Sphinx Manuals

You can include an example in reStructured Text as it is used by Sphinx with the `literalinclude` directive. This directive is used in this section to include the example above. The reST for this section is showed below the included example. Here is the example:

```
#include <WNS/Average.hpp>  
#include <WNS/TestFixture.hpp>  
  
namespace wns { namespace tests {  
    class AverageTest : public CppUnit::TestFixture {  
        TEST_SUITE( AverageTest );  
        TEST( testPutAndGet );  
        TEST( testReset );  
        TEST_SUITE_END();  
  
    public:  
        void setUp();  
        void tearDown();  
        void testPutAndGet();  
        void testReset();  
  
    private:  
        Average<double> average;  
    };  
}}
```

The reST source for this section is:

```
Including Examples in Sphinx Manuals  
-----
```

You can include an example in reStructured Text as it is used by Sphinx with the `literalinclude` directive. This directive is used in this section to include the example above. The reST for this section is showed below the included example. Here is the example:

```
.. literalinclude:: ../../../../createManualsWorkingDir/wns.avaragetest.header.example
```

The reST source for this section is:

## 5.2 Writing C++ Unit Tests

WNS uses `cppunit` for unit testing. The full manual is available at <http://cppunit.sourceforge.net/>. Here, you will be shown the basics of writing unit tests in openWNS to get you started quickly.

### 5.2.1 Quickstart

Let's have a look at a very simple unit test included in the `openwns-library`. `StopWatchTest.hpp` is the header file of this unit test.

```

1 #include <WNS/TestFixture.hpp>
2 #include <WNS/StopWatch.hpp>
3
4 namespace wns { namespace tests {
5
6     /**
7      * @brief test for wns::StopWatch
8      * @author Marc Schinnenburg <marc@schinnenburg.com>
9      */
10    class StopwatchTest :
11        public wns::TestFixture
12    {
13        TEST_SUITE( StopwatchTest );
14        TEST( testConstructor );
15            TEST( testGetInSeconds );
16        TEST_SUITE_END();
17
18    public:
19
20        void
21        prepare()
22        {
23        }
24
25        void
26        cleanup()
27        {
28        }
29
30        void
31        testConstructor()
32        {
33            Stopwatch sw;
34            ASSERT( 0.0 == sw.getInSeconds() );
35            ASSERT_MESSAGE( sw.toString(), sw.toString() == "0 s" );
36        }
37
38        void
39        testGetInSeconds()

```

```

40         {
41             Stopwatch sw;
42
43             // 2 s
44             timespec delay;
45             delay.tv_sec = static_cast<time_t>(2);
46             delay.tv_nsec = static_cast<long>(0*1E9);
47
48             sw.start();
49             nanosleep(&delay, NULL);
50             sw.stop();
51
52             ASSERT_EQUAL( 2, static_cast<int>(round(sw.getInSeconds())) );
53         }
54     };
55
56     TEST_SUITE_NAMED_REGISTRATION( StopwatchTest, wns::testsuite::Default() );
57
58 } // namespace tests
59 } // namespace wns

```

In line 1 we include the WNS TestFixture. The new class that implements our tests inherits from `wns::TestFixture` (cmp. line 11). You must use some cppunit macros to declare your test suite (`TEST_SUITE`, `TEST_SUITE_END`) and each test method (`TEST`). This is done in line 13-16.

If you need initialization or cleanup code you can place these in `prepare()` or `cleanup()`. `prepare()` is executed everytime *before* the test execution is passed to one of your test methods. `cleanup()` is executed everytime when test execution is returned to the test framework *after* your test method. If you have multiple test methods `prepare()` and `cleanup()` are called multiple times. This ensures that each test method sees the same clean environment when it starts. Individual tests do not have side effects on each other.

After that you can implement your test methods. These should be prefixed with `test`. Within your test you exercise your system under test and define your expectations by using the `ASSERT` macros.

**Note:** Include hyperlink to `ASSERT` overviews.

The most important line, which is often forgotten is line 56. `TEST_SUITE_NAMED_REGISTRATION` registers your test fixture with the test framework. If you forget this line your tests will not be executed. The first parameter is your test class, and the second defines to which test suite you register.

## 5.2.2 Why to use the openWNS Testfixture

Many units which are under test in WNS need an `EventScheduler` or `RandomNumberGenerator` to work. These elements (`EventScheduler`, `RNG`) are Singletons within the simulation, e.g. because the `EventScheduler` needs to be accessible from anywhere. Hence, the `EventScheduler` must be reset to be in a (defined) pristine state (no events pending, etc.) before a test is run. Similar things hold for the `RandomNumberGenerator`.

You should always use the `@c wns::TestFixture` in openWNS unit tests. It takes care of the proper initialization of the simulation platform and properly cleans up afterwards. `wns::TestFixture` is derived from `CppUnit::TestFixture` and implements `setUp()` and `tearDown()`.

To give the test writer the ability to prepare and cleanup his test too, the `wns::TestFixture` provides two methods: `prepare()` and `cleanup()`. The openWNS Testfixture implements the `setUp()` and `tearDown()` methods and takes care of initialization and resetting of simulator singletons. You can prepare and cleanup your test environment by implementing the `prepare()` and `cleanup()` methods.

### 5.2.3 Commonly used ASSERT Macros

The most simple assertion is to test if a boolean @em condition is true. If the @em condition is false during test execution the containing test fails and will be reported by the test environment.

```
CPPUNIT_ASSERT( boolean_condition)
```

Often you need to test if the **output** of your system under test meets your **expectation**, i.e. if the output value is equal to your expected value. You should use `CPPUNIT_ASSERT_EQUAL` for this. You could also use the simple `CPPUNIT_ASSERT` and include a comparison statement. However, the use of the `CPPUNIT_ASSERT_EQUAL` macro has the benefit that it includes the actual and expected value when reporting an error. You should **always** prefer `CPPUNIT_ASSERT_EQUAL` over `CPPUNIT_ASSERT` if you compare two values.

```
CPPUNIT_ASSERT_EQUAL( output , expectation )
```

Testing equality of `double` values is not reliable due to the floating point representation. You should test equality of two double values *output* and *expectation* by testing if  **$\left| \text{output} - \text{expectation} \right| < \text{maxAbsoluteError}$**

```
CPPUNIT_ASSERT_DOUBLES_EQUAL( output, expectation, maxAbsoluteError )
```

If you want to test for the relative error  **$\left| \frac{\text{output} - \text{expectation}}{\text{expectation}} \right| < \text{maxRelativeError}$**  you can use an openWNS extension of this macro

```
WNS_ASSERT_MAX_REL_ERROR( output, expectation, maxRelativeError )
```

**Note:** todo: Describe `WNS_ASSERT_ASSUME_EXCEPTION`, `WNS_ASSERT_ASSUME_NOT_NULL_EXCEPTION`, `CPPUNIT_ASSERT_MESSAGE`

```
CPPUNIT_ASSERT_MESSAGE( functionCall.getName(),
                        functionCall.getName() == "wns::Backtrace::snapshot()" );
```

### 5.2.4 Asserting that a certain Exception is thrown

Taken from `wns::AssureTest`

```
CPPUNIT_TEST_EXCEPTION( except, Assure::Exception );
```

Take from `container/tests/RegistryTest.cpp`:

```
CPPUNIT_ASSERT_THROW( r.update("foo", foo), AReg::UnknownKeyValue );
```

**Note:** todo: describe how to test that a certain exception is thrown

### 5.2.5 Controlling the Event Scheduler

When testing units that are part of an event driven simulation you somehow need to test if the behaviour in time of your system under test is correct. You need to have control over the progress of time in your test.

If the simulator executable is run in testing mode the event scheduler is initialized but the main eventloop is not started. The control on time progress is in the hands of the test developer. If you derive from `wns::TestFixture` the event scheduler is reset before entering your `prepare()` method.

In a test you would normally use the following code to control the event scheduler:

```
wns::simulator::getEventScheduler()->processOneEvent();
```

This tells the scheduler to execute exactly one event.

```
wns::simulator::Time now;
```

```
now = wns::simulator::getEventScheduler()->getTime();
```

This gives you the current time. You can use this to check if timeouts have been set correctly.

```
wns::simulator::getEventScheduler()->reset();
```

Resets the event scheduler. You probably won't use that, because @wns::TestFixture already executes this for you. However, you can always clear the event queue with this call.

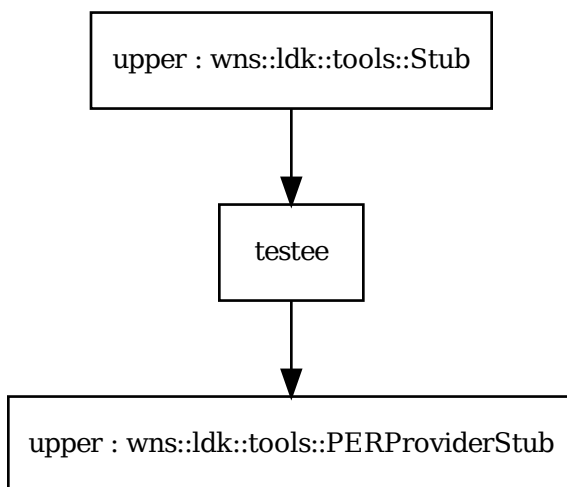
## 5.3 Writing Functional Units Tests

### 5.3.1 Setting up Functional Unit Networks

Whenever you need to test a functional unit you will need a test environment that provides a functional unit network for your testee. This example shows you how to setup a simple one and is taken from the `wns::ldk::crc::CRCTest`. You will need to include some headers for this to work:

```
#include <WNS/pyconfig/Parser.hpp>
#include <WNS/ldk/tests/LayerStub.hpp>
#include <WNS/ldk/fun/Main.hpp>
```

Now follows a walkthrough of the test setup of `wns::ldk::crc::CRCTest`. The setup constructs a simple FUN which is shown in the figure below.



First the following code is included in the `prepare` or `setUp` method.

```

1 void
2 CRCTest::setUp()
3 {
4     layer = new wns::ldk::tests::LayerStub();
5     fuNet = new fun::Main(layer);
6
7     wns::pyconfig::Parser emptyConfig;
8     upper = new tools::Stub(fuNet, emptyConfig);
9
10    ///BEWARE: Construction of this test's environment is not complete yet
11 } // setUp

```

This creates a stub for the layer, which is needed to construct a functional unit network (cmp. line 4 and 5). Make layer a member of your test and declare it as type `wns::ldk::ILayer*`.

**Warning:** Do not forget to delete `fun` and `layer` in your `tearDown` or `cleanup` method. Otherwise you will get a memory leak which shows up if you check the tests for leaks.

In line 7 an empty `wns::pyconfig::Parser` is constructed, which is essentially an empty `wns::pyconfig::View` which can be passed to your FU if it does not use config variables. Line 8 constructs the upper FU which is of type `wns::ldk::tools::Stub`. It is used to inject compounds to the functional unit network and inspect the incoming path. It does not require any configuration parameters

**Warning:** Do not delete any functional units in the `tearDown` or `cleanup` method of your test. Once a functional unit is constructed the functional unit network is responsible for the memory of its functional units. If you delete the FU after test execution this will result in a double delete.

Now let's have a look at the `setUpCRC` method of this test suite.

```

1 void
2 CRCTest::setUpCRC(const int _checksumSize, const bool _Dropping)
3 {
4     // Construct CRC FU
5     std::stringstream ss;
6     ss << "from openwns.CRC import CRC\n"
7        << "crc = CRC(\"PERstub\", \n"
8        << "  CRCsize = " << _checksumSize << ", \n"
9        << "  isDropping = "<< (_Dropping ? "True" : "False") << ") \n";
10
11    wns::pyconfig::Parser all;
12    all.loadString(ss.str());
13
14    wns::pyconfig::View config(all, "crc");
15
16    crc = new CRC(fuNet, config);
17 }

```

In line 5 to 12 the configuration of the testee is prepared. If you need to provide a config use the `loadString` method of `wns::pyconfig::Parser` which interprets the given string as Python code. It is a good idea to use the original `PyConfig` configuration class to setup your testee. In line 16 the CRC testee is constructed.

Now, the setup of the test environment is nearly complete. The CRC functional unit needs to access to a command which provides the packet error rate. We need to include a functional unit that provides this command in our test setup.

The next chapter describes in detail how the lower FU of type `PERProviderStub` is declared. To finish this setup let us have a look at the method `setUpPERProvider`.

```

1 void
2 CRCTest::setUpPERProvider(const double _PER)
3 {
4     // Construct fixed PER stub
5     std::stringstream ss;
6     ss << "fixedPER = " << _PER << "\n";
7
8     wns::pyconfig::Parser all;
9     all.loadString(ss.str());
10
11     lower = new PERProviderStub(fuNet, all);
12
13     upper
14         ->connect(crc)
15         ->connect(lower);
16
17     fuNet->addFunctionalUnit("upper", upper);
18     fuNet->addFunctionalUnit("myCRC", crc);
19     fuNet->addFunctionalUnit("PERstub", lower);
20     fuNet->onFUNCreated();
21
22     wns::simulator::getEventScheduler()->reset();
23
24 }

```

Line 1 to 11 us used to construct the `PERProviderStub` and its configuration. Then the FUs are connected according to the desired test environment setup. Afterwards, all FUs must be added to the functional unit network. Finally, the setup of the functional unit network is finalized by a call to `onFUNCreated`.

### 5.3.2 Satisfying a Command Dependency of your Testee

Suppose your testee needs to have a access to the command of another FU. The other FU however is quite complex and has a lot of dependencies which makes it infeasible to include that FU directly in your test. What you can do is create a stub for that FU that provides the proper command. Here is an example from the `PERProviderStub`. The command is defined as:

```

/**
 * @brief Provide a fixed PER for use by other layers
 *
 */
class PERProviderPCI :
    public EmptyCommand,
    public ErrorRateProviderInterface
{
public:
    PERProviderPCI()
        : PER(0.0)
        {};

    void
    setPER(double _PER) { PER = _PER; };

    virtual double
    getErrorRate() const { return PER; }
}

```

```
private:
    double PER;
}; // PERProviderPCI
```

However, it can be any command of your choice. You do not need derive from `EmptyCommand` or `ErrorRateProviderInterface`. Now what you need to do is create a Stub FU that provides this command in a simple functional unit network of your test. Within your test declare a class as follows:

```
class PERProviderStub :
    public StubBase,
    public CommandTypeSpecifier<PERProviderPCI>,
    public Cloneable<PERProviderStub>
```

It is a good idea to derive it from `wns::ldk::tools::StubBase` which provides control over accepting/wakeup functionality and records received and sent compounds. If this is used you need to implement the `calculateSizes` member method in your stub. Here is a minimal implementation.

```
// we need a unique overrider
virtual void
calculateSizes(const CommandPool* commandPool, Bit& commandPoolSize, Bit& dataSize) const
{
    StubBase::calculateSizes(commandPool, commandPoolSize, dataSize);
} // calculateSizes
```

### 5.3.3 Sending Compounds

Now that we have properly setup a test environment for a functional unit, we still need to test something. Let's have a look at the `testNoErrors` test of the `CRCTest`.

```
1 void
2 CRCTest::testNoErrors()
3 {
4     setUpCRC(checkSumSize, true);
5     setUpPERProvider(0.0);
6
7     upper->sendData(fuNet->createCompound());
8
9     ASSERT_EQUAL( size_t(1), lower->sent.size() );
10
11     lower->onData(lower->sent[0]);
12
13     ASSERT_EQUAL( size_t(1), upper->received.size() );
14
15 } // testNoErrors
```

First of all the test environment is setup. Remember that `setUp` or `prepare` is called by the testing framework. After this we call `setUpCRC` and `setUpPERProvider` to complete the setup.

In line 7 we tell the functional unit network to create an empty compound and then we use the `sendData` method of our upper FU to inject this compound. The upper FU will then try to pass it along the functional unit network calling `isAccepting` and `sendData` appropriately. Within this testcase we expect that the compound we pass to the network arrives at the lower FU. This is checked in line 9.

Then we use the `onData` method of the lower FU and pass on the compound in the incoming path. Here we also expect the compound to arrive at the upper FU. We check this in line 13.

## 5.4 Writing Python Unit Tests

openWNS uses the standard Python unit test framework.

### 5.4.1 Quickstart

Each of your test module can have an arbitrary name. However, it must be inside a directory called `tests`.

```
1 import unittest
2 import openwns.simulator
3
4 class ModuleMock:
5     def __init__(self, plugin):
6         self.__plugin__ = plugin
7
8 class ModulesTests(unittest.TestCase):
9
10    def setUp(self):
11        self.testee = openwns.simulator.Modules()
12        self.testee.foo = ModuleMock("moduleA")
13        self.testee.bar = ModuleMock("moduleB")
14
15    def testLen(self):
16        self.assertEqual(len(self.testee), 2)
17
18    def testAccess(self):
19        self.assertEqual(self.testee[0].__plugin__, "moduleA")
20        self.assertEqual(self.testee[1].__plugin__, "moduleB")
21        self.assertEqual(self.testee[-1].__plugin__, "moduleB")
22
23    def testIter(self):
24        foo = []
25        for it in self.testee:
26            foo.append(it)
27
28        self.assertEqual(foo[0].__plugin__, "moduleA")
29        self.assertEqual(foo[1].__plugin__, "moduleB")
30
31    #def testDenyDoubleAttributeAddition(self):
32    #    def provokeException():
33    #        self.testee.foo = ModuleMock("moduleC")
34    #        self.assertRaises(Exception, provokeException)
35
36    def testGetUnknownModule(self):
37        def provokeException():
38            tmp = self.testee.baz
39            self.assertRaises(Exception, provokeException)
40
41        try:
42            provokeException()
43        except Exception, e:
44            self.assertEqual("baz not available.\nAvailable modules are: foo, bar", str(e))
45
46    #def testModuleAlreadyRegistered(self):
47
48    #    def provokeException():
```

```
49     #         self.testee.baz = ModuleMock("moduleC")
50     #         self.testee.zab = ModuleMock("moduleC")
51     #     self.assertRaises(Exception, provokeException)
```



# CODING GUIDELINES

**Note:** For a short version please see `ref wns_documentation_codingguidelines_short`

## 6.1 Why do we need a style guide for openWNS?

This describes the coding conventions (guidelines, styles, rules, ... whatever you want) applying to the openWNS standard library and the main distribution. This style guide is inspired from “Style Guide for Python Code” (Python Enhancement Proposal: <http://www.python.org/dev/peps/pep-0008/>).

To give you an idea why these guidelines exist, we can again refer to “Style Guide for Python Code”:

```
One of Guido's key insights is that code is read much more often than it
is written. The guidelines provided here are intended to improve the
readability of code and make it consistent across the wide spectrum of
Python code. As PEP 20 says, "Readability counts".
```

```
A style guide is about consistency. Consistency with this style guide is
important. Consistency within a project is more important. Consistency
within one module or function is most important.
```

```
But most importantly: know when to be inconsistent -- sometimes the style
guide just doesn't apply. When in doubt, use your best judgment. Look
at other examples and decide what looks best. And don't hesitate to ask!
```

```
Two good reasons to break a particular rule:
```

- (1) When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
- (2) To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

## 6.2 Code layout

### 6.2.1 Indenting

**Use 4 spaces per indentation level.**

If tabs are used for indentation many editors mix tabs and spaces to get a nice indentation. This will result in an ugly indentation if the code is viewed with other settings for the tab-width. Example:

```
// edited with tab-width: 4

void
Foo::bar(int foo,
         string bar)
{
    // some code here
}
```

Since we have 9 characters in front of the type `string`, the editor set to tab-width 4 will use 2 tabs and one space to align `string` with `int`.

Opening this file in an editor with tab-width set to 8 looks like this:

```
// viewed with tab-width: 8

void
Foo::bar(int foo,
         string bar)
{
    // some code here
}
```

This is because two tabs is 8 spaces and one space add up to 17 spaces. This means tab-width is not adjustable. Thus the indentation style for openWNS is 4 spaces per indentation level.

### 6.2.2 Indentation in Emacs

The following shows a c-style for emacs that provides the required indentation rules for the openWNS.

```
;; The wns-c-style.
(defconst wns-c-style
  '( (c-tab-always-indent . t)
    (c-basic-offset . 4)
    (indent-tabs-mode . nil)
    (c-comment-only-line-offset . 4)
    (c-offsets-alist . ((comment-intro . 0)
                       (statement-block-intro . +)
                       (knr-argdecl-intro . +)
                       (substatement-open . 0)
                       (label . 0)
                       (statement-cont . +)
                       (inline-open . 0)
                       (inexpr-class . 0)
                       (inher-intro . ++)
                       ))
    ;; (c-echo-syntactic-information-p . t)
  )
  "WNS C/C++ Programming Style")

(c-add-style "WNS" wns-c-style)
```

Feel free to include the WNS c-style into you `~/.emacs` file.

## 6.2.3 Maximum line length

### Keep the line length readable

Suggestion is: around 80 characters or up to 10 words. Studies have shown that 10 words text-width are optimal for eye-tracking (from C++ Coding-Standards, Alexandrescu and Sutter, 2004).

## 6.2.4 Whitespaces

Conventional operators should be surrounded by a space character. C++ reserved words should be followed by a white space. Commas should be followed by a white space. Semicolons in for statements should be followed by a space character. Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <=, >=).

```
a = (b + c) * d;           // NOT: a=(b+c)*d
while (true)              // NOT: while(true) { ...
doSomething(a, b, c, d);  // NOT: doSomething(a,b,c,d);
for (int ii = 0; ii < 10; ++ii) // NOT: for(i=0;i<10;i++)
```

Avoid extraneous whitespace in the following situations:

Immediately inside parentheses, brackets or braces. Immediately before the open parenthesis that starts the argument list of a function call. Immediately before the open parenthesis that starts an indexing. To align operators with others.

```
spam(ham, eggs(2))       // NOT: spam( ham, eggs (2) )
spam(1)                  // NOT: spam (1)
map['key'] = vector[index] // NOT: map ['key'] = vector [index]
int a = 0;               // NOT: int a    = 0
int b = 0;               // NOT: int b    = 0
int index = 0;          // NOT: int index = 0
```

## 6.2.5 Namespaces

```
namespace a { namespace b { namespace c {
} // namespace c
} // namespace b
} // namespace a
```

## 6.2.6 Classes and Templates

```
template <typename FOO, typename BAR>
class FooBar :
    public virtual SuperClassA
    private SuperClassB
{
public:
    // Public stuff first (most interesting)
protected:
    // Then protected stuff (only interesting if deriving)
private:
```

```
    // Last private stuff (internal realization)
};
```

## 6.2.7 Methods

```
// of course a virtual template method is impossible
// but the keywords should be placed like this
template <typename FOO, typename BAR>
virtual const BAR&
scheduleFooBar(const FOO& foo) const
{
    // some code
}
```

## 6.2.8 Constructors

```
// in case you have only one argument don't forget to add
// explicit keyword to avoid automatic type cast
explicit
FooBar::FooBar(Arg arg) :
    SuperClass(arg),
    argument(arg)
{
    // no member initialization here!
}
```

## 6.2.9 Control structures (if, while, for)

```
if (this->empty() == true)
{
    // some code here
}
else
{
    // some code here
}

while (this->empty() == false)
{
    // some code here
}

for (int ii = 0; ii < 4; ++ii)
{
    // some code here
}
```

## 6.2.10 Type modifiers

\* and & belong to the type.

```
int* a = new a;    // NOT: int *a = new a; or int * a = new a;
int& b = c;       // NOT: int &b = c; or int & b = c;
```

## 6.2.11 Comments

Comments are placed above the code block to be commented. Usage of trailing comments is explicitly discouraged:

```
// Yes:
// Make sure the station got registered
if(std::find(stationContainer.begin(), stationContainer.end(), station))
{
    station->move();
}

// No:
if(std::find(stationContainer.begin(), stationContainer.end(), station)) // Make sure the station got
{
    station->move();
}
```

Don't write comments that just repeat the code. They will likely get out of sync.

```
// No:
if(!container.empty()) // true if container not empty
{
    container.clear();
}
```

## 6.3 Naming Conventions

Names are always written in CamelCase style. **Not** in underscore\_style. The only exceptions are defines (include guards / macros).

### 6.3.1 Local variables

Variables start lower case, even if they start with an acronym. The acronym is all written in lower case then:

```
void
someMethod()
{
    double someVariable;
    Station umtsStation;
}
```

## 6.3.2 Methods

Methods start lower case. They describe an action. Therefore, they should contain a verb:

```
void  
sendData(const PDU& pdu)  
{  
}
```

For non-virtual interfaces (NVI)s the method to actually dispatch the request is prefixed with do:

```
void  
sendData(const PDU& pdu)  
{  
    this->doSendData(pdu);  
}
```

```
virtual void  
doSendData(const PDU& pdu) = 0;
```

Methods, that represent (asynchronous) event-based interfaces are prefixed on:

```
void  
onConnectionEstablished()  
{  
}
```

And for an event-based NVI:

```
void  
onConnectionEstablished()  
{  
    this->doOnConnectionEstablished();  
}  
virtual void  
doOnConnectionEstablished() = 0;
```

## 6.3.3 Classes

Classes start with an upper case letter and are written in CamelCase:

```
class PositionProvider  
{  
};
```

If a part of the name is an acronym, the acronym is written in upper case letters (or in mixed case if this is the normal way of spelling it):

```
class UMTSTransmitter  
{  
};
```

```
class WiMAXReceiver  
{  
};
```

### 6.3.4 Interfaces

Interfaces start with an `I`. This avoids name collisions as it is quite common to have `IFoo` and a class named `Foo` that implements `IFoo`.

```
class IComponent
{
public:
    virtual void
        connect() = 0;
};
```

### 6.3.5 Class Members

Scope identification of a variable is important. Is it a local scratch variable or a class member. For easy identification of class members a underscore at the end of the member is used, regardless if the member is private, protected or public:

```
class Foo
{
private:
    int bar_;
};
```

### 6.3.6 Namespaces

Namespace are written all in lowercase letters:

```
namespace wns { namespace simulator {
}
}

using namespace wns::simulator;
```

### 6.3.7 Template Parameters

Template parameters are written in upper case letters:

```
template <typename KEY, typename VALUE>
class Registry
{
}
```

### 6.3.8 Macros

Macros are written in upper case letters. Underscores should be used for better readability. The name should always begin with `WNS_` to avoid collisions with other macros (e.g. `REF` and `DEREF` of Qt).

```
#define WNS_ADD(x, y) (x)+(y)
```

### 6.3.9 Include Guards

Include guards, like Macros, are written in upper case letters with underscores to enhance readability. To avoid name clashes here the following rule must be followed when choosing a name for an include guard:

```
MODULE_DIR_SUBDIR_SUBSUBDIR_FILE
```

Hence, if the file is placed in the TCP module in `src/congestion/Tahoe.hpp` the include guard is:

```
#ifndef TCP_CONGESTION_TAHOE_HPP
#define TCP_CONGESTION_TAHOE_HPP
// some code
#endif // NOT defined TCP_CONGESTION_TAHOE_HPP
```

At the closing `#endif` you should state what is (not) defined here.

## 6.4 Programming recommendations

### 6.4.1 Namespaces

Some rules: `#`. Never use `using namespace xyz` in header files `#`. For an implementation: Only use `using namespace xyz` for the corresponding class definition `#`. You should omit the additional namespace qualifiers if you are already in that namespace (e.g. in a class definition) `#`. You should always use the full namespace qualifier for any out-of-current-namespace-scope types

```
namespace foo {
    class Bar
    {
        void
        clone(Bar*);
    };
}

// NOT
namespace foo {
    class Bar
    {
        void
        clone(foo::Bar*);
    };
}
```

### 6.4.2 Comparison

When testing for something (e.g. in an if-statement), always be explicit about what you expect:

```
// No
if (foo)

// Yes (because it can be any of these)

// in case of bool
if (foo == true)
```

```
// in case of pointer
if (foo != NULL)

// in case of integer
if (foo != 0)
```

### 6.4.3 Call-by-value, call-by-reference

Use call by reference (const) where possible for complex data types, but always call-by-value for Plain Old Data Types (PODs):

```
// YES
void
foo(const Bar& bar)

// NO
void
foo(Bar bar)

// NO, only use if you need to modify bar inside foo! and even then with care!
void
foo(Bar& bar)

// NO, only use if you need polymorphism
void
foo(Bar* bar)

// YES
void
foo(double x)

// NO
void
foo(const double& x)
```

## 6.5 License Statement

openWNS is released under the Lesser GNU Public License Version 2. We follow the guidelines of the Free Software Foundation for releasing software under the LGPLv2 (see <http://www.gnu.org/licenses/gpl-howto.html> for details).

This recommendation requires to put a Header at the beginning of every released file that states that this source code is part of openWNS, that it is released under the GPLv2 and that it comes with no warranty. Below you find a prepared header both for Python and C++. Every module should also contain a copy of the GNU Public License and the Lesser GNU Public License. Please place the files <http://www.openwns.org/Wiki/CodingStyles?action=AttachFile&do=view&target=COPYING> and <http://www.openwns.org/Wiki/CodingStyles?action=AttachFile&do=view&target=COPYING.LESSER> in the root of your module.

The LGPLv2 Header for C++ to be used in openWNS is as follows:

```
/*
 * *****
 * This file is part of openWNS (open Wireless Network Simulator)
 * _____
 */
```

```
*
* Copyright (C) 2004-2007
* Chair of Communication Networks (ComNets)
* Kopernikusstr. 16, D-52074 Aachen, Germany
* phone: ++49-241-80-27910,
* fax: ++49-241-80-22242
* email: info@openwns.org
* www: http://www.openwns.org
*
* _____
*
* openWNS is free software; you can redistribute it and/or modify it under the
* terms of the GNU Lesser General Public License version 2 as published by the
* Free Software Foundation;
*
* openWNS is distributed in the hope that it will be useful, but WITHOUT ANY
* WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
* A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more
* details.
*
* You should have received a copy of the GNU Lesser General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*****/
```

The LGPLv2 Header for Python to be used in openWNS is as follows:

```
#####
# This file is part of openWNS (open Wireless Network Simulator)
#
# _____
#
# Copyright (C) 2004-2007
# Chair of Communication Networks (ComNets)
# Kopernikusstr. 16, D-52074 Aachen, Germany
# phone: ++49-241-80-27910,
# fax: ++49-241-80-22242
# email: info@openwns.org
# www: http://www.openwns.org
#
# _____
#
# openWNS is free software; you can redistribute it and/or modify it under the
# terms of the GNU Lesser General Public License version 2 as published by the
# Free Software Foundation;
#
# openWNS is distributed in the hope that it will be useful, but WITHOUT ANY
# WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
# A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more
# details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
#
#####
```

# CODING GUIDELINES REFERENCE CARD

## 7.1 Layout Rules

Topic	Rule
Indenting	spaces, not tabs
Maximum line length	not fixed (although above 100 is considered long)

## 7.2 Naming Conventions

Names are always written in CamelCase style. Not in `underscore_style`. The only exceptions are defines (include guards / macros).

Topic	Rule	Example
Local variables	Variables start lower case, even if they start with an acronym.	<code>double someVariable Station umtsStation</code>
Methods	Methods start lower case. They describe an action. Therefore, they should contain a verb. NVI is prefixed with <code>do</code> , event-based interface with <code>on</code>	<code>void sendData() void doSendData() void onData()</code>
Classes	Classes start with an upper case letter and are written in CamelCase	<code>class PositionProvider</code>
Interfaces	Interfaces start with an I	<code>class IComponent</code>
Class members	End with an underscore	<code>int bar_</code>
Namespaces	Namespace are written all in lowercase letters	<code>namespace wns using namespace wns::simulator</code>
Template Parameters	Template parameters are written in upper case letters	<code>template &lt;typename KEY, typename VALUE&gt;</code>
Macros	Macros are written in upper case letters. Underscores should be used for better readability.	<code>#define WNS_ADD(x, y) (x)+(y)</code>
Include Guards	Include guards, like Macros, are written in upper case letters with underscores to enhance readability	<code>MODULE_DIR_SUBDIR_SUBSUBDIR_FILE #ifndef CP_CONGESTION_TAHOE_HPP</code>

# DEBUGGING YOUR CODE

## 8.1 Finding Memory leaks

To find memory leaks you can use valgrind. Reasonable settings are. Replace ROOTOFSDK and YOUR\_PROGRAM

```
> valgrind --tool=memcheck --leak-check=yes --num-callers=20 --leak-resolution=high \  
  --log-file=val.log --suppressions=ROOTOFSDK/config/valgrind.supp YOUR_PROGRAM
```

## 8.2 CPU Profiling

To CPU profile your program you can also use valgrind. Use this commands:

```
> sconsc optassuremsg callgrind=True  
> valgrind --tool=callgrind --instr-atstart=no ROOTOFSDK/sandbox/callgrind/bin/openwns  
> kcachegrind callgrind.out.*
```

Alternatively you can use gprof to perform CPU profiling.

```
> sconsc optassuremsg profile=True
```

Afterwards use gprof to postprocess the profiling results.

## 8.3 SmartPtr Debugging

If you have cyclic SmartPtrs you probably want to make use of the SmartPtr debugging functionality of openwns. Execute

```
> sconsc dbg smartPtrDBG=True  
> ROOTOFSDK/sandbox/smartptrdbg/openwns
```

At the end of the simulation run all SmartPtrs that were not properly deleted are shown. Each occurrence is accompanied by the call stack that led to its construction.

**Note:** Please note that it is possible that openwns quits with a SIGSEGV at the simulation end. This is due to the undeterministic destruction sequence of static variables and the way SmartPtr debugging works.

# TODO LIST

**Todo**

This is another todo entry

(The original entry is located in `writingCode/WritingCode.rst`, line 5 and can be found [here](#).)



# INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*



# INDEX

## E

event scheduler, 19  
events, 19

## F

finite state machine, 40  
FSM, 40

## M

measurement source, 25  
measurements  
    measurement source, 25

## S

scheduler, 19  
state machine, 40