FW4SPL Documentation

Release 0

IRCAD-IHU

October 23, 2015

Contents

CHAPTER 1

Installation

1.1 Latest Release Version

The last release of the project is:

- fw4spl_0.10.1

1.2 Installation for Windows

1.2.1 Prerequisites for Windows users

If not already installed:

1. Install Mercurial
2. Optionally you can install TortoiseHg
3. Install Visual Studio 2013
4. Install Python 2.7
5. Install CMake
6. Install jom
7. Install ninja

Qt is an external library used in FW4SPL. For the successful compilation of Qt with FW4SPL, please see the following requirements:

- http://wiki.qt.io/Building_Qt_5_from_Git

1.2.2 FW4SPL installation

FW4SPL works with data separation for source, build and install data. To prepare the development environment:

- Create a development folder (Dev)
- Create a build folder (Dev\Build)
- Create a source folder (Dev\Src)
- Create a install folder (Dev\Install)

To prepare the third party environment:

- Create a third party folder (BinPkgs)
- Create a build folder (BinPkgs\Build)
- Create a source folder (BinPkgs\Src)
- Create a install folder (BinPkgs\Install)

Dependencies

For the third party libraries the three following repositories have to be cloned in the (BinPkgs) source folder:

- fw4spl-deps
- fw4spl-ar-deps
- fw4spl-ext-deps

Update the cloned repositories to the used version. Make sure that cmake is set as environment variable. Call the cmake-gui or change the cmake arguments with the console from the BinPkgs build folder location. Choose jom as build tool (make sure jom was set as path variable to be found by cmake) for cmake. The following arguments have to be changed:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps
- *CMAKE_INSTALL_PREFIX*: set the install location.

Configure and generate the code. Compile the FW4SPL dependencies with jom in the console (e.g. jom all, jom qt, etc).

Source

For the FW4SPL source code the three following repositories have to be cloned in the (Dev) source folder:

- fw4spl
- fw4spl-ar
- fw4spl-ext

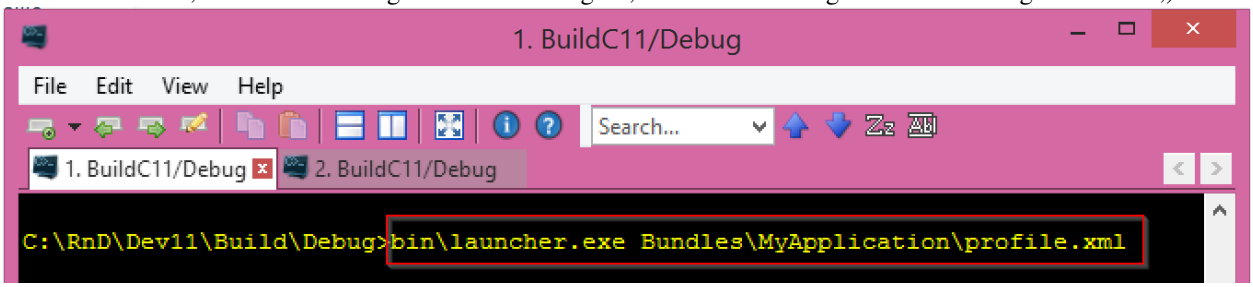
Update the cloned repositories to the used version. Make sure that cmake is set as environment variable. Call the cmake-gui or change the cmake arguments with the console from the Dev build folder location. Choose jom or ninja as build tool for cmake (make sure the build tool was set as path variable to be found by cmake). The following arguments have to be changed:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext
- *CMAKE_INSTALL_PREFIX*: set the install location.
- *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.

Make sure the arguments concerning the compiler (advanced arguments) point to Visual Studio. Configure and generate the code. Compile the FW4SPL source code with jom or ninja in the console . To develop applications with FW4SPL the source code can be imported and compiled with the preferred development environment.

1.2.3 Launch an application

To work with an specific application or several applications the cmake argument *PROJECTS_TO_BUILD* can be set. After an successful compilation the application can be launched with the launcher.exe from FW4SPL. Therefore the profile.xml of the application in the build folder has to be passed as argument to the launcher call in the console. Make sure that the external libraries directory is set to the path (set PATH=%PATH%;C:FW4SPLBinPkgsInstallPathDebugbin;C:FW4SPLBinPkgsInstallPathDebugx64vc12bin;). .



1.2.4 Recommended software

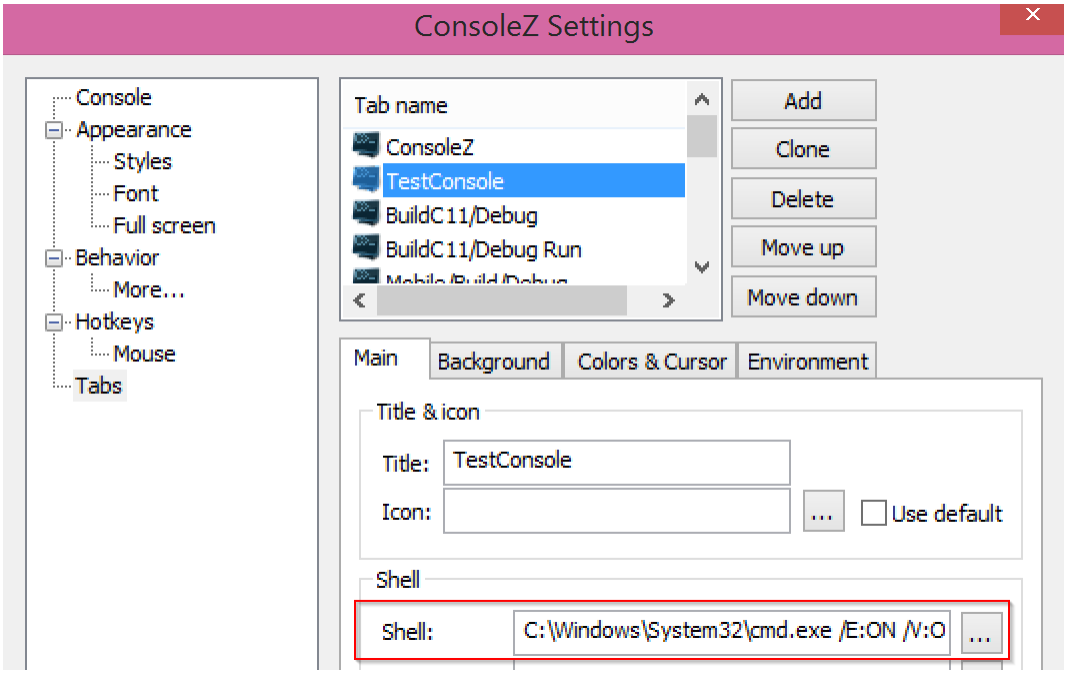
The following programs may be helpful for your developments:

- Install Eclipse CDT. Eclipse is a multi-OS Integrated Development Environment (IDE) for computer programming.
- Install Notepad++. Notepad++ is a free source code editor, which is designed with syntax highlighting functionality.
- Install ConsoleZ. ConsoleZ is an alternative command prompt for Windows, adding more capabilities to the default Windows command prompt. To compile FW4SPL with the console the windows command prompt has to be set in the tab settings.

1.2.5 Console settings

To use FW4SPL with the ConsoleZ or Console2 the following settings should be done:

- The current directory should point to the build directory
- Notify the windows command prompt as shell (e.g. C:WindowsSystem32cmd.exe /E:ON /V:ON /T:0E /K), as default in debug mode
- Join .bat files to the shell settings, to indicate specific settings like path extensions to the external libraries of FW4SPL (e.g. set PATH=%PATH%;C:FW4SPLBinPkgsInstallPathDebugbin;C:FW4SPLBinPkgsInstallPathDebugx64vc12bin;)



1.2.6 Release

To generate the projects in release, the following instruction has to be added:

- The console shell should be the windows command prompt in release mode (C:WindowsSystem32cmd.exe /E:ON /V:ON /T:0E /K /release)
- Change CMake argument CMAKE_BUILD_TYPE to release
- Reference the EXTERNAL_LIBRARIES to the install folder of third part libraries compiled in release mode (for compiling the FW4SPL projects)

1.3 Installation for Linux

1.3.1 Prerequisites for Linux users

If not already installed:

1. Install Mercurial (apt-get install mercurial)
2. Optionally you can install TortoiseHg
3. Install gcc (apt-get install gcc)
4. Install clang (apt-get install clang)
4. Install Python 2.7 (apt-get install python2.7)
5. Install CMake (apt-get install cmake)
6. Install jom (apt-get install jom)

Qt is an external library used in FW4SPL. For the successful compilation of Qt with FW4SPL, please see the following requirements:

- http://wiki.qt.io/Building_Qt_5_from_Git

1.3.2 FW4SPL installation

FW4SPL works with data separation for source, build and install data. To prepare the development environment:

- Create a development folder (Dev)
- Create a build folder (Dev\Build)
- Create a source folder (Dev\Src)
- Create a install folder (Dev\Install)

To prepare the third party environment:

- Create a third party folder (BinPkgs)
- Create a build folder (BinPkgs\Build)
- Create a source folder (BinPkgs\Src)
- Create a install folder (BinPkgs\Install)

Dependencies

For the third party libraries the three following repositories have to be cloned in the (BinPkgs) source folder:

- fw4spl-deps
- fw4spl-ar-deps
- fw4spl-ext-deps

Update the cloned repositories to the used version. Call the cmake-gui or change the cmake arguments with the console from the BinPkgs build folder location. Choose jom or nmake as build tool for cmake. The following arguments have to be changed:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps
- *CMAKE_INSTALL_PREFIX*: set the install location.

Configure and generate the code. Compile the FW4SPL dependencies with jom or nmake in the console.

Source

For the FW4SPL source code the three following repositories have to be cloned in the (Dev) source folder:

- fw4spl
- fw4spl-ar
- fw4spl-ext

Update the cloned repositories to the used version. Make sure that cmake is set as environment variable. Call the cmake-gui or change the cmake arguments with the console from the Dev build folder location. Choose jom or nmake as build tool for cmake. The following arguments have to be changed:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext
- *CMAKE_INSTALL_PREFIX*: set the install location.
- *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.

Make sure the arguments concerning the compiler (advanced arguments) point to the preferred compiler. Configure and generate the code. Compile the FW4SPL source code with jom or nmake in the console . To develop applications with FW4SPL the source code can be imported and compiled with the preferred development environment.

1.3.3 Launch an application

To work with an specific application or several applications the cmake argument *PROJECTS_TO_BUILD* can be set. After an successful compilation the application can be launched with the launcher program from FW4SPL. Therefore the profile.xml of the application in the build folder has to be passed as argument to the launcher call in the console. (bin/launcher Bundles/MyApplicationAndVersion/profile.xml)

1.3.4 Recommended software

The following programs may be helpful for your developments:

- Install Eclipse CDT. Eclipse is a multi-OS Integrated Development Environment (IDE) for computer programming.

1.3.5 Release

To generate the projects in release, the following instruction has to be added:

- Change CMake argument CMAKE_BUILD_TYPE to release
- Reference the EXTERNAL_LIBRARIES to the install folder of third part libraries compiled in release mode (for compiling the FW4SPL projects)

1.4 Installation for MacOSX

1.4.1 Prerequisites for MacOSX users

If not already installed:

1. Install Mercurial
2. Optionally you can install TortoiseHg
4. Install Python 2.7
5. Install CMake

Tip:

1. You can also install Ninja instead of using **make**.
2. For an easy install, you can use the Hombrew project to install missing packages.

1.4.2 FW4SPL installation

FW4SPL works with data separation for source, build and install data. To prepare the development environment:

- Create a development folder (Dev)

- Create a build folder (Dev\Build)
- Create a source folder (Dev\Src)
- Create a install folder (Dev\Install)

To prepare the third party environment:

- Create a third party folder (BinPkgs)
- Create a build folder (BinPkgs\Build)
- Create a source folder (BinPkgs\Src)
- Create a install folder (BinPkgs\Install)

Build tools

FW4SPL is a CMake project. That means, for each build target there is a CMakeLists that provides build parameters. To configure you project you can use the `cmake` command from the build folder with the sources as arguments:

```
ccmake /PATH/TO/fw4spl-deps
```

if you want to use **Ninja** as build to tools, use the option `-G Ninja`, as following:

```
ccmake -G Ninja /PATH/TO/fw4spl-deps
```

It is the same process for BinPkgs and FW4SPL sources.

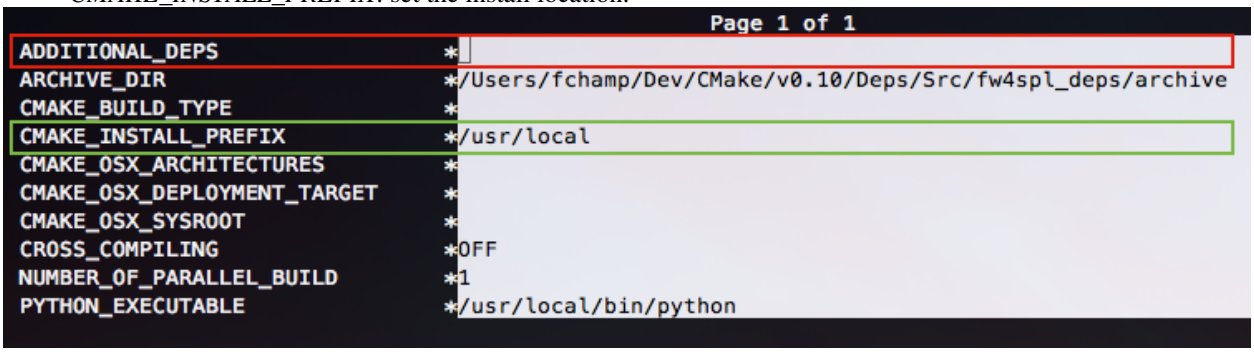
Dependencies

For the third party libraries the three following repositories have to be cloned in the (BinPkgs) source folder:

- fw4spl-deps
- fw4spl-ar-deps
- fw4spl-ext-deps

To build dependencies see *Build tools* instructions. Some CMake variables have to be change:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps
- *CMAKE_INSTALL_PREFIX*: set the install location.



Press configure (*[c]*) and generate (*[g]*) makefiles. Then, compile the FW4SPL dependencies with make or ninja in a terminal.

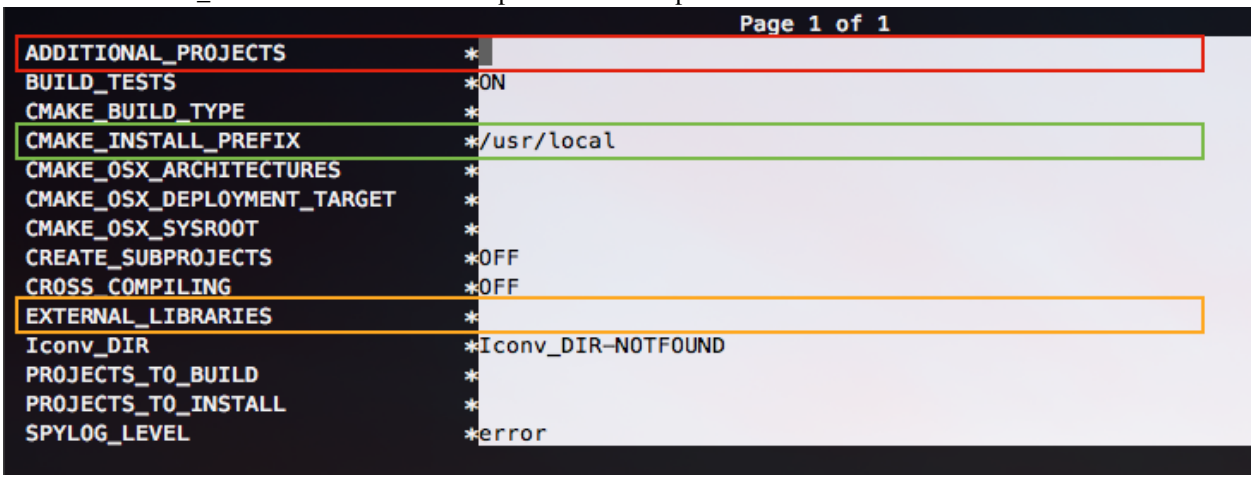
Source

For the FW4SPL source code the three following repositories have to be cloned in the (Dev) source folder:

- fw4spl
- fw4spl-ar
- fw4spl-ext

To build soruces see *Build tools* instructions. Some CMake variables have to be change:

- *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext
- *CMAKE_INSTALL_PREFIX*: set the install location.
- *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.



Press configure (**[c]**) and generate (**[g]**) makefiles. Then, build dependencies with make or ninja in a terminal.

example: `make Qt` or `ninja Qt`

1.4.3 Launch an application

To build a specific or several applications the CMake argument `PROJECTS_TO_BUILD` can be set.

Tip: Use `;` so separate each application name.

After an successful compilation the application can be launched with the launcher program from a terminal. Therefore the profile.xml of the application in the build folder has to be passed as argument to the launcher:

```
bin/launcher Bundles/MyApplicationAndVersion/profile.xml)
```

Note: To generate the projects in release, the following instruction has to be change:

- Change CMake argument `CMAKE_BUILD_TYPE` to release
- Set the `EXTERNAL_LIBRARIES` to the release install folder of dependencies

1.4.4 Recommended softwares

The following programs may be helpful for your developments:

- **IDE:**
  - Qt creator
  - Eclipse CDT.
- **Versioning tools:**
  - **'TortoiseHg <http://tortoisehg.bitbucket.org/'_**
  - **'SourceTree <http://www.sourcetreeapp.com/'_**

CHAPTER 2

Software Architecture Description (SAD)

2.1 General

2.1.1 Introduction

The framework FW4SPL (FrameWork for Software Production) is an open-source framework, developed by IRCAD (research institute against cancer and disease). The principle of FW4SPL is the fast and easy creation of applications, mainly in the medical field. Therefore it provides features like digital image processing in 2D and 3D, visualization or simulation of medical interactions. To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

FW4SPL is built on component-based architecture composed of C++ libraries. The three main concepts of the architecture, explained in the following sections, are:

- object-service concept
- component approach
- signal-slot communication

The framework is multi-platform and runs under Windows, Linux and MacOS. The programming language of the framework is C++. This document will introduce the general architecture of FW4SPL.

2.1.2 Annexes

- *Srclib list:* this document lists all libraries with a brief description.
- *Object list:* this document lists all data with a brief description.
- *Service list:* this document lists all services and bundles with a brief description.
- *Third party:* this document contains a description of libraries used to support this architecture and its functions.
- *OSR diagram:* this document introduces how to represent an application configuration as a diagram.

2.2 Object-Service concept

2.2.1 Introduction

Inside the Object Oriented Programming (OOP) paradigm, an object is an instance of a class which contains all its data and methods (such as reading, writing, visualizations, image analysis, etc.). This philosophy works well, provided that

the classes do not change with time. However, in this situation, the maintenance of source code is difficult.

In order to make this maintenance easier, FW4SPL architecture relies on an Object-Service paradigm where data and their methods are separated into different code units.

2.2.2 Object

Objects represent data used in the framework. They can be simple (boolean, integer, string, etc.) or advanced structures (image, mesh, video, patient, etc.) without depending on the input data format. For example, an input image could have one of several formats such as Jpeg or Dicom but the FW4SPl object will be the same.

Moreover, these object classes contain only data features and their corresponding getter/setter methods.

For instance, the `Image` object:

- contains a buffer pointer, a buffer size, the image's dimension and origin,
- has public setter/getter methods to access these members,
- does not have methods such as reading or writing a buffer

The `fwData` library contains the standard simple and advanced data. It is the FW4SPL's main data library. There is also the `fwMedData` library which contains several structures to store medical data. A data list with a brief description is available in the appendixes.

Creating data

New data must be created as described below.

In the header file (MyData.hpp):

```
class MyData : public ::fwData::Object
{

public :
    fwCoreClassDefinitionsWithFactoryMacro( (MyData)(::fwData::Object),
        (()), ::fwData::factory::New< MyData >) ;


    // Private constructor, required for data factory
    MyData(::fwData::Object::Key key);

    /// Destructor, required for all data
    virtual ~MyData();

    /// Defines shallow copy, required for all data
    void shallowCopy( const Object::csptr& _source );

    /// Defines deep copy, required for all data
    void cachedDeepCopy(const Object::csptr& _source,
                            DeepCopyCacheType &cache);

};
```

In the source file (MyData.cpp), this line must also be added to declare `MyClass` as data of the framework architecture :

```
fwDataRegisterMacro( MyData );
```

2.2.3 Service

A service represents a functionality which uses or modifies data. A service is always associated with a data. For example, image data can have a reader service, a writer service, a visualization service or a processing operator.

Service type

Some service categories exist in FW4SPL. These categories are called *service types* and are represented by an abstract class. The basic service types are:

- `io::IReader`: base interface for reader services.
- `io::IWriter`: base interface for writer services.
- `fwGui::IActionSrv`: base interface to manage action from a button or a menu in the GUI.
- `gui::editor::IEditor`: base interface to create new widget in the GUI.
- `fwRender::IRender`: base interface to create new visualization widgets in the GUI.
- `fwServices::IController`: Does nothing in particular but can be considered as a default service type to be implemented by unclassified services.

All services require a type association and must inherit from an abstract type service.

Service methods

Several methods exist to manipulate a service. The main methods are: `configure`, `start`, `stop`, `update` and `receive`.

- `configure`: parses the service parameters and analyze its configuration. For example, this method is used to configure an image file path on the file system for an image reader service.
- `start`: initializes and launch the service (be careful, starting and instantiating a service is not the same thing. For example, for a visualization service, the `start` method instantiates all GUI widgets necessary to visualize the data but the service itself is instantiated before.).
- `stop`: stops the service. For example, for a visualization service, this method detaches and destroys all GUI widgets previously instantiated earlier in the `start` method.
- `update` method is called to perform an action on the data associated with the service. For example, for an image reader service, the service reads the image, converts it and loads it into the associated data.
- `receive` is called when the service associated object is modified. The method parameter contains all the information about this modification. For example, after an image object update has been realized by an image reader service, the associated image visualization service is notified that the image buffer has been modified and then, the view is refreshed.

This method is mandatory, but can be empty. This is because some services do not need a start/stop process, an update process or to listen to object modifications.

Service states

These methods must follow a calling sequence. For example, it is not possible to stop a service before starting it. To secure the process, a state machine has been implemented to control the calling sequence.

The calling sequence to manage a service is:

```
MyData::sptr myData = MyData::New();
MyService::sptr mySrv = MyService::New();
mySrv->setObject(myData);

mySrv->setConfiguration( ... ); // set parameters
mySrv->configure(); // check parameters
mySrv->start(); // start the service
mySrv->update(); // update the service
mySrv->stop(); // stop the service
```

Create a service

A new service must be created as described below.

In the header file (MyService.hpp):

```
class MyService : public AbstractServiceType
{
public:

    // Macro to define few important parameters/functions
    // used by the architecture
    fwCoreServiceClassDefinitionsMacro((MyService)(AbstractServiceType));

    // Service constructor
    MyService() throw() ;

    // Service destructor.
    virtual ~MyService() throw() ;

protected:

    // To configure the service
    void configuring() throw(fwTools::Failed);

    // To start the service
    void starting() throw(::fwTools::Failed);

    // To stop the service
    void stopping() throw(::fwTools::Failed);

    // To receive notification about object modification
    void receiving( CSPTR(::fwServices::ObjectMsg) _msg )
                    throw(::fwTools::Failed);

    // To update the service
    void updating() throw(::fwTools::Failed);
};
```

In the source file (MyService.cpp), this line must be also added to declare `MyService` as a service of the framework architecture:

```
fwServicesRegisterMacro( AbstractServiceType, MyService, MyData );
```

Note: When a new service is created, the following functions must be overloaded from IService class : `configuring`, `starting`, `stopping`, `receiving` and `updating`. The top level functions from IService class checks the service state before any call to the redefined method.

2.2.4 Object and service factories

To instantiate an object or a service, the architecture requires the use of a factory system. In class-based programming, the factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating classes without specifying the exact class that will be created. This is done by creating classes via a factory method, which is either specified in an interface (abstract class) and implemented in implementing classes (concrete classes) or implemented in a base class (optionally as a template method), which can be overridden when inherited in derivative classes; rather than by a constructor[#]_.

Object factory

The `fwData` library has a factory to register and create all objects. The registration is managed by two macros:

```
// in .hpp file
fwCoreClassDefinitionsWithFactoryMacro( (MyData)(::fwData::Object),
    ((), ::fwData::factory::New< MyData >);

// in .cpp file
fwDataRegisterMacro( MyData );
```

Then, there are only two ways to build data in the framework:

```
// Direct creation
MyData::sptr obj = MyData::New();

// Factory creation (here obj is an object of type
// MyData, it is possible to cast it)
::fwData::Object::sptr obj = ::fwData::factory::New("MyData");
```

Service factory

The `fwService` library has a factory to register and create all services. The registration is managed by two macros:

```
// in .hpp file
fwCoreServiceClassDefinitionsMacro ((MyService)(AbstractServiceType));

// in .cpp file
fwServicesRegisterMacro( AbstractServiceType, MyService, MyData );
```

Then, there is only one way to build a service in the framework:

```
::fwServices::registry::ServiceFactory::sptr srvFactory
        = ::fwServices::registry::ServiceFactory::getDefault();

// Factory creation (here srv is a service of type MyService,
// it is possible to cast it)
::fwServices::IService::sptr srv = srvFactory->create("MyService");
```

2.2.5 Object-Service registry

The FW4SPL architecture is standardized thanks to:

- Abstract classes `fwData::Object` and `fwService::IService`.

- The two factory systems.

In an application, one of the problems is managing the life cycle of a large number of object instances and their services. This problem is solved by the class `fwServices::registry::ObjectService` which maintains the relationship between objects and services. This class concept is very simple :

```
// OSR is a singleton
class ObjectService
{
  // relation map beetwen an object and his associated services
  map < Object *, vec < IService > > osr;

  // Associates a service to an object
  // manages in the function the association: srv->setObject(obj);
  void registerService ( Object * obj , IService * srv );

  // Dissociates a service to his object
  void unregisterService ( IService * srv );

  // ...
}

// Some helpers exist : below, add method is used to combine
// factory system with service registration
::fwServices::IService::sptr add(::fwData::Object::sptr obj,
        std::string serviceType, std::string _implementationId)
```

This registry manages the object-service relationships and guarantees the non-destruction of an object while some services are still working on it.

2.2.6 Object-Service concept example

To conclude, the generic object-service concept is illustrated with this example:

```
// Create an object
::fwData::Object::sptr obj = ::fwData::factory::New("::fwData::Image");

// Create a reader and a view for this object
::fwServices::IService::sptr reader
    = ::fwServices::add(obj, "::io::IReader", "MyCustomImageReader");
::fwServices::IService::sptr view
    = ::fwServices::add(obj, "::fwRender::IRender", "MyCustomImageView");

// Configure and start services
reader->setConfiguration ( /* ... */ );
reader->configure();
reader->start();

view->setConfiguration ( /* ... */ );
view->configure();
view->start();

// Execute services
reader->update(); // Read image on filesystem
view->update(); // Refresh visualization with the new image buffer

// Stop services
reader->stop();
```

```
view->stop();

// Destroy services
::fwServices::registry::ObjectService::unregisterService(reader);
::fwServices::registry::ObjectService::unregisterService(view);
```

This example shows the code to create a small application to read an image and visualize it. You can easily transform this code to build an application which reads and displays a 3D mesh by changing object and services implementation strings only.

2.3 Signal-slot communication

2.3.1 Overview

"Signals and slots" is a language construct introduced in Qt [1] for communication between objects.

The concept is that objects and services(explained in 2.3) can send signals containing event information which can be received by other services using special functions known as slots.

2.3.2 FW4SPL implementation

In the FW4SPL architecture, the library `fwCom` provides a set of tools dedicated to communication. These communications are based on the signal and slot concept.

`fwCom` provides the following features :

- function and method wrapping
- direct slot calling
- asynchronous slot calling
- ability to work with multiple threads
- auto-disconnection of slot and signals
- arguments loss between slots and signals

2.3.3 Slots

Slots are wrappers for functions and class methods that can be attached to a `fwThread::Worker`. The purpose of this class is to provide synchronous and asynchronous mechanisms for method and function calling.

Slots have a common base class : SlotBase. This allows the storage of them in the same container. Slots are designed such that they can be called, even where only the argument type is known.

Examples :

A slot wrapping the function sum, which is a function with the signature int (int, int) :

```
::fwCom::Slot< int (int, int) >::sptr slotSum = ::fwCom::newSlot( &sum );
```

A slot wrapping the function start with signature void() of the object `a` which class type is `A` :

[1] http://wiki.qt.io/Qt_signal-slot_quick_start

```
::fwCom::Slot< void() >::sptr slotStart = ::fwCom::newSlot(&A::start, &a);
```

Execution of the slots using the run method :

```
slotSum->run(40,2);
slotStart->run();
```

Execution of the slots using the method call, which returns the result of the execution :

```
int result = slotSum->call(40,2);
slotStart->call();
```

A slot declaration and execution, through a SlotBase :

```
::fwCom::Slot< size_t (std::string) > slotLen
        = ::fwCom::Slot< size_t (std::string) >::New( &len );
::fwCom::SlotBase::sptr slotBaseLen = slotLen;
::fwCom::SlotBase::sptr slotBaseSum = slotSum;
slotBaseSum->run(40,2);
slotBaseSum->run<int, int>(40,2);

// This one needs the  explicit argument type
slotBaseLen->run<std::string>("R2D2");
result = slotBaseSum->call<int>(40,2);
result = slotBaseSum->call<int, int, int>(40,2);
result = slotBaseLen->call<size_t, std::string>("R2D2");
```

2.3.4 Signals

Signals allow to perform grouped calls on slots. For this purpose, a signal class provides a mechanism to connect slots.

Examples:

The following instruction declares a signal with a void signature.

```
::fwCom::Signal< void() >::sptr sig = ::fwCom::Signal< void() >::New();
```

The connection between a signal and a slot of the same information type:

```
sig->connect(slotStart);
```

The following instruction will trigger the execution of all slots connected to this signal:

```
sig->emit();
```

It is possible to connect multiple slots with the same information type to the same signal and trigger their simultaneous execution.

Signals can take several arguments as a signature which will trigger their connected slots by passing the right arguments.

In the following example a signal is declared of type void(int, int). The signal is connected to two different types of slot, void (int) and int (int, int).

```
using namespace fwCom;
Signal< void(int, int) >::sptr sig2 = Signal< void(int, int) >::New();
Slot< int(int, int) >::sptr    slot1 = Slot< int(int, int) >::New(...);
Slot< void(int) >::sptr        slot2 = Slot< void(int) >::New(...);

sig2->connect(slot1);
```

```
sig2->connect(slot2);

sig2->emit(21, 42);
```

Here 2 points need to be highlighted :

- A signal cannot return a value. Consequently their return type is void. Thus, the return value of a slot, triggered by a signal, equally cannot be retrieved.
- To successfully trigger a slot using a signal, the minimum requirement as to the number of arguments or fitting argument types has to be given by the signal. In the last example the slot slot2 only requires one argument of type int, but the signal is emitting two arguments of type int. Because the signal signature fulfills the slot's argument number and argument type, the signal can successfully trigger the slot slot2. The slot slot2 takes the first emitted argument which fits its parameter (here 21, the second argument is ignored).

Disconnection

The disconnect method is called between one signal and one slot, to stop their existing connection. A disconnection assumes a signal slot connection. Once a signal slot connection is disconnected, it cannot be triggered by this signal. Both connection and disconnection of a signal slot connection can be done at any time.

```
sig2->disconnect(slot1);
sig2->emit(21, 42); // do not trigger slot1 anymore
```

The instructions above will cause the execution of slot2. Due to the disconnection between sig2 and slot1, the slot slot1 is not triggered by sig2.

Connection handling

The connection between a slot and a signal returns a connection handler:

```
::fwCom::Connection connection = signal->connect(slot);
```

Each connection handler provides a mechanism which allows a signal slot connection to be disabled temporarily. The slot stays connected to the signal, but it will not be triggered while the connection is blocked :

```
::fwCom::Connection::Blocker lock(connection);
signal->emit();
// 'slot' will not be executed while 'lock' is alive or until lock is
// reset
```

Connection handlers can also be used to disconnect a slot and a signal :

```
connection.disconnect();
// slot is not connected anymore
```

Auto-disconnection

Slots and signals can handle an automatic disconnection :

- on slot destruction : every signal slot connection to this slot will be destroyed
- on signal destruction : every slot connection to the signal will be destroyed

All related connection handlers will be invalidated when an automatic disconnection occurs.

2.3.5 Manage slots or signals in a class

The library `fwCom` provides two helper classes to manage signals or slots in a structure.

HasSlots

The class `HasSlots` offers mapping between a key (string defining the slot name) and a slot. `HasSlots` allows the management of many slots using a map. To use this helper in a class, the class must inherit from `HasSlots` and must register the slots in the constructor:

```
struct ThisClassHasSlots : public HasSlots
{
  typedef Slot< int()> GetValueSlotType;

  ThisClassHasSlots()
  {
      GetValueSlotType::sptr slotGetValue
            = ::fwCom::newSlot( &SlotsTestHasSlots::getValue, this );
      HasSlots::m_slots("sum", &SlotsTestHasSlots::sum, this)
                       ("getValue", slotGetValue );
  }

  int sum(int a, int b)
  {
      return a+b;
  }

  int getValue()
  {
      return 4;
  }
};
```

Then, slots can be used as below :

```
ThisClassHasSlots obj;
obj.slot("sum")->call<int>(5,9);
obj.slot< ThisClassHasSlots::GetValueSlotType >("getValue")->call();
```

HasSignals

The class `HasSignals` provides mapping between a key (string defining the signal name) and a signal. `HasSignals` allows the management of many signals using a map, similar to `HasSlots`. To use this helper in a class, the class must inherit from `HasSignals` as seen below and must register signals in the constructor:

```
struct ThisClassHasSignals : public HasSignals
{
  typedef ::fwCom::Signal< void()> SignalType;

  ThisClassHasSignals()
  {
      SignalType::sptr signal = SignalType::New();
      HasSignals::m_signals("sig", signal);
  }
};
```

Then, signals can be used as below:

```
ThisClassHasSignals obj;
Slot< void()>::sptr slot = ::fwCom::newSlot(&anyFunction)
obj.signal("sig")->connect( slot );
obj.signal< SignalsTestHasSignals::SignalType >("sig")->emit();
obj.signal("sig")->disconnect( slot );
```

2.3.6 Signals and slots used in objects and services

Slots are used in both objects and services, whereas signals are only used in services. The abstract class `fwData::Object` inherits from the `HasSignals` class as a basis to use signals :

```
class Object : public ::fwCom::HasSignals
{
  /// Key in m_signals map of signal m_sigObjectModified
  static const ::fwCom::Signals::SignalKeyType s_OBJECT_MODIFIED_SIG;

  /// Type of signal m_sigObjectModified
  typedef ::fwCom::Signal< void ( CSPTR( ::fwServices::ObjectMsg ) ) >
               ObjectModifiedSignalType;

  /// Signal that emits an ObjectMsg when an object is modified
  ObjectModifiedSignalType::sptr m_sigObjectModified;

  Object()
  {
      m_sigObjectModified = ObjectModifiedSignalType::New();
      m_signals( s_OBJECT_MODIFIED_SIG,  m_sigObjectModified);
  }
}
```

Moreover the abstract class `fwService::IService` inherits from the `HasSlots` class and the `HasSignals` class, as a basis to communicate through signals and slots:

```
class IService : public ::fwCom::HasSlots, public ::fwCom::HasSignals
{
  /// Key in m_slots map of slot m_slotReceive
  static const ::fwCom::Slots::SlotKeyType s_RECEIVE_SLOT;

  /// Type of signal m_slotReceive
  typedef ::fwCom::Slot<void(ObjectMsg::csptr)> ReceiveSlotType;

  /// Slot to call receive method
  ReceiveSlotType::sptr m_slotReceive;

  IService()
  {
      m_slotReceive  = ::fwCom::newSlot( &IService::receive   , this ) ;
      ::fwCom::HasSlots::m_slots( s_RECEIVE_SLOT , m_slotReceive )
  }
}
```

According to the design, the `s_OBJECT_MODIFIED_SIG` object signal is connected to all `s_RECEIVE_SLOT` slots of its associated services (object service relation). When a service modifies its associated object, the service emits an `s_OBJECT_MODIFIED_SIG` signal from the object in order to notify any service working on the modified object through the receive method.

An other way to communicate between objects and services is to split each modification type into different signals and to create different slots in the services. In this case, the method `IService::getObjSrvConnections()` and the helper `::fwServices::helper::SigSlotConnection` provide few tools to connect/disconnect signals/slots between objects/services.

2.3.7 Proxy

The class `::fwServices::registry::Proxy` is a communication element and singleton in the architecture. It defines a proxy for signal/slot connections. The proxy concept is used to declare communication channels: all signals registered in a proxy's channel are connected to all slots registered in the same channel. This concept is useful to create multiple connections or when the slots/signals have not yet been created (possible in dynamic programs).

The following shows an example where one signal is connected to several slots:

```
const std::string CHANNEL = "myChannel";

::fwServices::registry::Proxy::sptr proxy
    = ::fwServices::registry::Proxy::getDefault();

::fwCom::Signal< void() >::sptr sig = ::fwCom::Signal< void() >::New();

::fwCom::Slot< void() >::sptr slot1 = ::fwCom::newSlot( &myFunc1 );
::fwCom::Slot< void() >::sptr slot2 = ::fwCom::newSlot( &myFunc2 );
::fwCom::Slot< void() >::sptr slot3 = ::fwCom::newSlot( &myFunc3 );

proxy->connect(CHANNEL, sig);

proxy->connect(CHANNEL, slot1);
proxy->connect(CHANNEL, slot2);
proxy->connect(CHANNEL, slot3);

sig->emit(); // All slots are called
```

2.3.8 Object messages

The communication system called *communication channel system* used in the former versions of FW4SPL, was replaced by the signal slot communication system. As a result of this replacement, object messages were introduced. With each object modification, a message is sent informing services that an object modification has occurred. The signals and slots use a message parameter to store information about the object modification or to specialize the message from others. The library `fwComEd` contains all message structures which can be used to communicate object modifications. As shown in the table below, several messages are available for each object.

Objects	Available messages
Acquisition	{ADD_RECONSTRUCTION, VISIBILITY, NEW_RECONSTRUCTION_SELECTED}
Boolean	{VALUE_IS_MODIFIED}
Camera	{NEW_CAMERA, CAMERA_MOVING}
Color	{VALUE_IS_MODIFIED}
Composite	{MODIFIED_FIELDS, ADDED_FIELDS, REMOVED_FIELDS, SWAPPED_FIELDS}
Float	{VALUE_IS_MODIFIED}
Graph	{NEW_GRAPH, ADD_NODE, REMOVE_NODE, ADD_EDGE, REMOVE_EDGE, SELECTED_NODE, UNSELECTED_NODE, ...}
Image	{NEW_IMAGE, BUFFER, MODIFIED, DIMENSION, SPACING, REGION, PIXELTYPE, TRANSFERTFUNCTION, ...}
Integer	{VALUE_IS_MODIFIED}
Interaction	{MOUSE_LEFT_UP, MOUSE_RIGHT_UP, MOUSE_MIDDLE_UP, MOUSE_WHEELFORWARD_UP, MOUSE_WHEELBACKWARD_UP, ...}
Location	{LOCATION_IS_MODIFIED}
Material	{MATERIAL_IS_MODIFIED}
Model	{NEW_MODEL}
PatientDB	{NEW_PATIENT, ADD_PATIENT, CLEAR_PATIENT, NEW_IMAGE_SELECTED, NEW_LOADED_PATIENT, NEW_RESECTION_SELECTED}
Patient	{NEW_PATIENT, NEW_MATERIAL_FOR_RECONSTRUCTION}
PlaneList	{ADD_PLANE, REMOVE_PLANE, PLANELIST_VISIBILITY, PLANELIST_MODIFIED, DESELECT_ALL_PLANES}
Plane	{PLANE_MODIFIED, START_PLANE_INTERACTION, DESELECT_PLANE, WAS_SELECTED, WAS_DESELECTED}
PointList	{ELEMENT_MODIFIED, ELEMENT_ADDED, ELEMENT_REMOVED}
Point	{POINT_IS_MODIFIED, START_POINT_INTERACTION}
Reconstruction	{MESH, VISIBILITY}
ResectionDB	{NEW_RESECTIONDB_SELECTED, RESECTIONDB_SELECTED, NEW_RESECTION_SELECTED, NEW_SAFE_PART_SELECTED, ...}
Resection	{ADD_RECONSTRUCTION, VISIBILITY, NEW_RECONSTRUCTION_SELECTED, MODIFIED}
Spline	{NEW_SPLINE}
String	{VALUE_IS_MODIFIED}
Tag	{TAG_IS_MODIFIED}
...	...

2.4 App-config

2.4.1 Dynamic program with factories

As shown in the *Object-Service concept example*, it is easy to change data and service to modify the application behavior by working on a mesh instead of an image. However, this is limited to one service working with one data. It is impossible to manage several objects/services to create complex software.

Then FW4SPL architecture provides a dynamic management of configurations to allow the use of multiple objects and services.

2.4.2 Dynamic program with application configuration

In the `fwService` library, an application configuration parser allows to parse XML files then creates and manages objects, services and communications.

```
// The parser
void main (int argc , char * argv [])
{
    string xmlAppConfigPath = argv [1];

    ::fwServices::AppConfigManager::sptr acm
            = ::fwServices::AppConfigManager::New();

    acm->setConfig(xmlAppConfigPath);
    acm->create(); // Creates objects and services from config.
    acm->start(); // Starts services specified in config.
    acm->update(); // Updates services specified in config.

    acm->stop(); // Stops services specified in config.
    acm->destroy(); // Destroy all services and then data.
}
```

The following part correspond to the configuration XML file of the previous *Object-Service concept example*.

```
<object uid="image" type ="::fwData::MyData">

    <service uid="frame" impl="DefaultFrame" type="IFrame" >
        <!-- service configuration -->
    </service>

    <service uid="view" impl="MyCustomImageView"
             type="::fwRender::IRender" >
        <!-- service configuration -->
    </service>

    <service uid="reader" impl="MyCustomImageReader"
             type="::io::IReader" >
        <!-- service configuration -->
    </service>

    <!-- view listen now image modification -->
    <connect>
        <signal>image/objectModified</signal>
        <slot>view/receive</slot>
    </connect>

    <start uid="frame" />
    <start uid="view"/>
    <start uid="reader"/>

    <!-- Read the image on filesystem and notify
         the view to refresh is content -->
    <update uid ="reader"/>

</ object >
```

This is a simple example to show how to build an application with several objects and services thanks to a program and its configurations files.

2.5 Activities

An activity represents a set of services. This set can be used as a sub part of an application or as an application.

The SActivityLauncher service allows to launch an activity. Its role is to create the specific Activity associated with the selected data.

- *::activities::action::SActivityLauncher* uses the selected data to generate the activity.
- *::guiQt::editor::DynamicView* displays the activity in the application.
- *Vector* contains the set of selected data .

2.6 Multithreading

2.6.1 Overview

The multithreading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge to prominence from the more specialized field of transaction processing:

- Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs.
- Techniques that would allow speedup of the overall system throughput of all tasks would be a meaningful performance gain.

Some advantages include:

- If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed.
- If a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread can avoid leaving these idle.
- If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization of its values.

Some criticisms of multithreading include:

- Multiple threads can interfere with each other when sharing hardware resources such as caches or translation look aside buffers (TLBs).
- Execution times of a single thread are not improved but can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.
- Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing.
- Thread scheduling is also a major problem in multithreading.

Michael K. Gschwind, et al. [2]

[2] Michael K. Gschwind, Valentina Salapura. 2011. Using Register Last Use Infomation to Perform Decode-Time Computer Instruction Optimization US 20130086368 A1 [Patent]. http://www.google.com/patents/US20130086368

2.6.2 Worker and Timer

In the FW4SPL architecture, the library `fwThread` provides few tools to execute asynchronous tasks on different threads.

In this library, the class `Worker` creates and manages a task loop. The default implementation creates a loop in a new thread. Some tasks can be posted on the worker and will be executed on the managed thread. When the worker is stopped, it waits for the last task to be processed and stops the loop.

```
::fwThread::Worker::sptr worker = ::fwThread::Worker::New();

::boost::packaged_task<void> task( ::boost::bind( &myFunction ) );
::boost::future< void > future = task.get_future();
::boost::function< void () > f = moveTaskIntoFunction(task);

worker->post(f);

future.wait();
worker->stop();
```

The Timer class provides single-shot or repetitive timers. A Timer triggers a function once after a delay, or periodically, inside the worker loop. The delay or the period is defined by the duration attribute.

```
::fwThread::Worker::sptr worker = ::fwThread::Worker::New();

::fwThread::Timer::sptr timer = worker->createTimer();

timer->setFunction(  ::boost::bind( &myFunction)  );

::boost::chrono::milliseconds duration
        = ::boost::chrono::milliseconds(100) ;
timer->setDuration(duration);

timer->start();
//...
timer->stop();

worker->stop();
```

2.6.3 Mutex

The namespace `fwCore::mt` provides common foundations for multithreading in FW4SPL, especially tools to manage mutual exclusions. In computer science, mutual exclusion refers to the requirement of ensuring that two concurrent threads are not in a critical section at the same time, it is a basic requirement in concurrency control, to prevent race conditions. Here, a critical section refers to a period when the process accesses a shared resource, such as shared memory. A lock system is designed to enforce a mutual exclusion concurrency control policy.

Currently, FW4SPL uses Boost Thread library which allows the use of multiple execution threads with shared data, keeping the C++ code portable. `fwCore::mt` defines a few typedef over Boost:

```
namespace fwCore
{
namespace mt
{

typedef ::boost::mutex Mutex;
typedef ::boost::unique_lock< Mutex > ScopedLock;
```

```
typedef ::boost::recursive_mutex RecursiveMutex;
typedef ::boost::unique_lock< RecursiveMutex > RecursiveScopedLock;

/// Defines a single writer, multiple readers mutex.
typedef ::boost::shared_mutex ReadWriteMutex;
/**
* @brief Defines a lock of read type for read/write mutex.
* @note Multiple read lock can be done.
*/
typedef ::boost::shared_lock< ReadWriteMutex > ReadLock;
/**
* @brief Defines a lock of write type for read/write mutex.
* @note Only one write lock can be done at once.
*/
typedef ::boost::unique_lock< ReadWriteMutex > WriteLock;
/**
* @brief Defines an upgradable lock type for read/write mutex.
* @note Only one upgradable lock can be done at once but there
        may be multiple read lock.
*/
typedef ::boost::upgrade_lock< ReadWriteMutex > ReadToWriteLock;
/**
* @brief Defines a write lock upgraded from ReadToWriteLock.
* @note Only one upgradable lock can be done at once but there
        may be multiple read lock.
*/
typedef ::boost::upgrade_to_unique_lock< ReadWriteMutex >
            UpgradeToWriteLock;

} //namespace mt
} //namespace fwCore
```

2.6.4 Multithreading and communication

Asynchronous call

Slots are able to work with `fwThread::Worker`. If a Slot has a Worker, each asynchronous execution request will be run in its worker, otherwise asynchronous requests can not be satisfied without specifying a worker.

Setting worker example:

```
::fwCom::Slot< int (int, int) >::sptr slotSum
        = ::fwCom::newSlot( &sum );
::fwCom::Slot< void () >::sptr slotStart
        = ::fwCom::newSlot( &A::start, &a );

::fwThread::Worker::sptr w = ::fwThread::Worker::New();
slotSum->setWorker(w);
slotStart->setWorker(w);
```

`asyncRun` method returns a boost::shared_future< void >, that makes it possible to wait for end-of-execution.

```
::boost::future< void > future = slotStart->asyncRun();
// do something else ...
future.wait(); //ensures slotStart is finished before continuing
```

`asyncCall` method returns a `boost::shared_future< R >` where R is the return type. This allows facilitates waiting for end-of-execution and retrieval of the computed value.

```
::boost::future< int > future = slotSum->asyncCall();
// do something else ...
future.wait(); //ensures slotStart is finished before continuing
int result = future.get();
```

In this case, the slots asynchronous execution has been *weakened*. For an async call/run pending in a worker queue, it means that :

- if the slot is detroyed before the execution of this call, it will be canceled.
- if slot's worker is changed before the execution of this call, it will also be canceled.

Asynchronous emit

As slots can work asynchronously, triggering a Signal with asyncEmit results in the execution of connected slots in their worker :

```
sig2->asyncEmit(21, 42);
```

The instruction above has the consequence of running each connected slot in its own worker.

Note: Each connected slot must have a worker set to use asyncEmit.

2.6.5 Object-Service and Multithreading

Object

The architecture allows the writing of thread safe functions which manipulate objects easily. Objects have their own mutex (inherited from `fwData::Object`) to control concurrent access from different threads. This mutex is available using the following method:

```
::fwCore::mt::ReadWriteMutex & getMutex();
```

The namespace `fwData::mt` contains several helpers to lock objects for multithreading:

- `ObjectReadLock`: locks an object mutex on read mode.
- `ObjectReadToWriteLock`: locks an object mutex on upgradable mode.
- `ObjectWriteLock`: locks an object mutex on exclusive mode.

The following example illustrates how to use these helpers:

```
::fwData::String::sptr m_data = ::fwData::String::New();
{
    // lock data to write
    ::fwData::mt::ObjectReadLock readLock(m_data);
} // helper destruction, data is no longer locked


{
    // lock data to write
    ::fwData::mt::ObjectWriteLock writeLock(m_data);

    // unlock data
    writeLock.unlock();
```

```
    // lock data to read
    ::fwData::mt::ObjectReadToWriteLock updrageLock(m_data);

    // unlock data
    updrageLock.unlock();

    // lock again data to read
    updrageLock.lock();

    // lock data to write
    updrageLock.upgrade();

    // lock data to read
    updrageLock.downgrade();

} // helper destruction, data is no longer locked
```

Services

The service architecture allows the writing of a thread-safe service by avoiding the requirement of explicit synchronization. Each service has an associated worker in which service methods are intended to be executed.

Specifically, all inherited `IService` methods (`start`, `stop`, `update`, `receive`, `swap`) are slots. Thus, the whole service life cycle can be managed in a separate thread.

Since services are designed to be managed in an associated worker, the worker can be set/updated by using the inherited method :

```
// Initializes m_associatedWorker and associates
// this worker to all service slots
void setWorker( ::fwThread::Worker::sptr worker );

// Returns associate worker
::fwThread::Worker::sptr getWorker() const;
```

Since the signal-slot communication is thread-safe and `IService::receive(msg)` method is a slot, it is possible to attach a service to a thread and send notifications to execute parallel tasks.

Note: Some services use or require GUI backend elements. Thus, they can't be used in a separate thread. All GUI elements must be created and managed in the application main thread/worker.

2.7 Serialization

2.7.1 Overview

Serialization is the process to save plain C++ structures from memory to hard drive. In fw4spl, `fwAtoms` library provides tools to serialize all data (and especially Object that extend `::fwData::Object`) to a JSON format [3]. Of course, this process is also available for loading data from JSON format to plain C++ structures.

To achieve this serialization, `fwAtoms` provides basic structures (which extend `::fwAtoms::Base`) to manage better plain C++ structure evolution. Thus, there are two main steps in the serialization process:

[3] Introducing JSON. http://json.org/

- Converting a `::fwData::Object` into a `::fwAtoms::Object`
- Serializing a `::fwAtoms::Base` in a JSON format

2.7.2 Atom objects

The basic structures provided by `fwAtoms` library are a set of restricted C++ type. All these structures extend `::fwAtoms::Base` and cover all basic types and containers:

type	brief
::fwAtoms::Base	Base class of all atoms
::fwAtoms::String	Atom to represent string types
::fwAtoms::Numeric	Atom to represent numeric types (floating number or integer)
::fwAtoms::Boolean	Atom to represent a boolean value
::fwAtoms::Map	Atom to represent an associative container (std::string to ::fwAtoms::Base)
::fwAtoms::Sequence	Atom to represent a sequence of object like vector or list
::fwAtoms::Object	Atom to represent a C++ object with attributes
::fwAtoms::Blob	Atom to represent binary information like buffers

For instance, consider the following C++ class:

```
class SimpleClass
{
    bool m_myBoolean;
};

class ComplexClass
{
    std::string m_myString;
    float m_myFloat;
    SimpleClass* m_mySimpleClass;
};
```

It's Atom equivalent is (simplified code):

```
fwAtoms::Object
{
    metaInfos
    {
        "CLASSNAME_METAINFO" : "SimpleClass"
        "ID_METAINFO" : "<ID of the object>"
    }

    attributes
    {
        "myBoolean" : ::fwAtoms::Boolean
    }
}


fwAtoms::Object
{
    metaInfos
    {
        "CLASSNAME_METAINFO" : "ComplexClass"
        "ID_METAINFO" ; "<ID of the object>"
    }
```

```
    attributes
    {
        "myString" : ::fwAtoms::String
        "myFloat" : ::fwAtoms::Numeric
        "mySimpleClass" : ::fwAtoms::Object("SimpleClass")
    }
}
```

The main advantage of this representation is the ability to change easily the form of a class. In fact, all plain C++ objects are represented as `Atoms::Object` with a map of attributes. Thus, adding, removing or changing the content of an attribute is easy. Moreover, because of these restricted types, atom parsing is also made easier. The main difficulty is how to convert plain C++ object using this set of restricted types.

2.7.3 Convert a `fwData::Object`

As explained earlier, all objects in fw4spl inherit from the `::fwData::Object` class. To convert a C++ object in Atom, it must inherit from this class. To allow this conversion, some work must be done.

The first thing is to update the header file of the structure and add these lines :

```
// Before all namespace
fwCampAutoDeclareDataMacro((<namespace elem>)
        (<namespace elem>)(<class name>), <method export macro>);

// In the public class part
fwCampMakeFriendDataMacro((<namespace elem>)
        (<namespace elem>)(<class name>));
```

These two functions allow the declaration of the class to the conversion process.

Next, the conversion systems must know the class information including attributes, base class, library location and data version. This is achieved by creating a class which defines these properties.

Example

This can be illustrated by taking the previous class and creating these two files:

Header file of the newly created class: ComplexClass.hpp

```
// Reference class

fwCampAutoDeclareDataMacro((fwData)(ComplexClass), FWDATA_API);

namespace fwData
{
class ComplexClass : public ::fwData::Object
{
    fwCampMakeFriendDataMacro((fwData)(ComplexClass));

    std::string m_myString;
    float m_myFloat;
    ::fwData::SimpleClass* m_mySimpleClass;
};
}
```

Header file of serialization class :

```
// hpp binding file
#include <fwCamp/macros.hpp>
#include <fwData/ComplexClass.hpp>
#include "fwDataCamp/config.hpp"

fwCampDeclareAccessor((fwData)(ComplexClass), (fwData)(SimpleClass));
```

Source file of serialization class :

```
// cpp binding file
// include previous cpp file

#include <fwCamp/UserObject.hpp>

fwCampImplementDataMacro((fwData)(ComplexClass))
{
    builder
        .tag("object_version", "1")
        .tag("lib_name", "fwData")
        .base< ::fwData::Object>()
        .property("myString" , &::fwData::ComplexClass::m_myString)
        .property("myFloat" , &::fwData::ComplexClass::m_myFloat)
        .property("mySimpleClass" , &::fwData::ComplexClass::m_mySimpleClass)
        ;
}
```

In a header file, the method fwCampDeclareAccessor is necessary when an object has a pointer or a smart pointer to another object.

In a source file, fwCampImplementDataMacro declares the properties of the bound object with an object called a builder: it provides several methods to describe the object to bind.

method	brief
tag(key, value)	Register a tag in the atom meta information.
base<BaseClass>()	Identify the base class of the bound object
property(arg1, arg2)	Set property of the object and how to access it

Most of the work is completed when the header file of the relevant class has been updated and a binding class created. The last step is to register the binding class in the conversion system using the following line in the library containing binding classes:

```
localDeclarefwDataComplexClass();
```

In fw4spl, data are located in `fwData` library whereas data binding classes are located in `fwDataCamp` library. The above line registering a binding class can be found in `fwDataCamp` autoload.hpp files.

Serialization file example

For more information about serialization see:

location	brief
Srclib/core/fwData/include/	fwData header files with serialization macros
Srclib/core/fwDataCamp	Serialization description of all fw4spl data
Srclib/core/fwDataCamp/include/fwDataCamp/autoload.hpp	Auto loading data bindings in the system

`fwData::Object` to `fwAtoms::Object` conversion

The requirements to convert an `fwData::Object` into an `fwAtoms::Object` are in the `fwAtomConversion` library.

Two functions are necessary to achieve this conversion:

```
//Convert a fwData::Object into fwAtoms::Object
SPTR(::fwAtoms::Object) convert( const SPTR(::fwData::Object) &data );

//Convert a fwAtoms::Object into fwData::Object
SPTR(::fwData::Object) convert( const SPTR(::fwAtoms::Object) &atom );
```

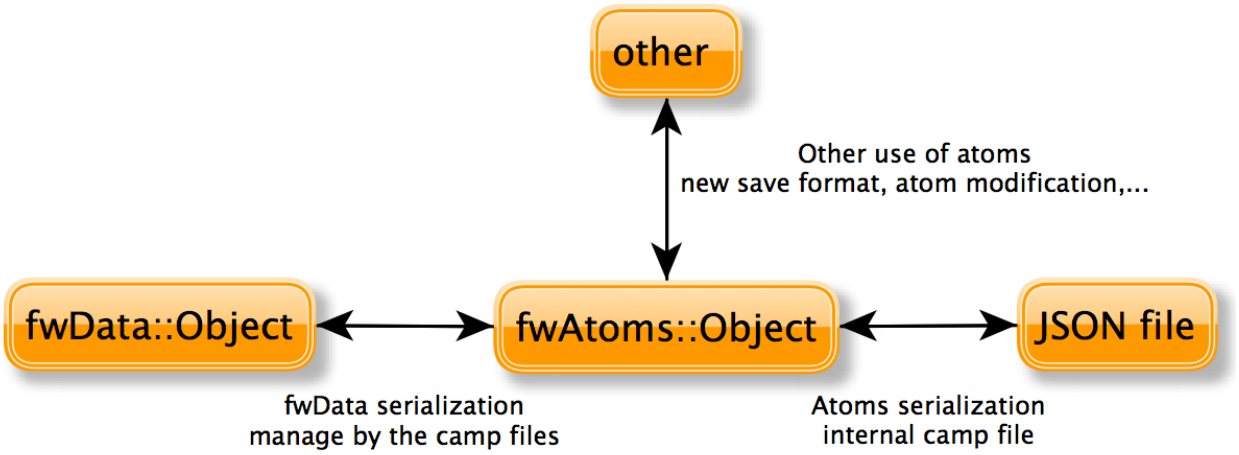
2.7.4 Serialize an Atoms object to JSON format

When a fw4spl data is converted into Atoms, it can be saved in JSON format. Both an Atom reader and Atom writer are available in the `fwAtomsBoostIO` fw4spl library: simply instantiate one of these classes with an Atom object and call the read or write method.

To serialize atoms into JSON, a visitor pattern is used. An example can be found in the `fwAtomsBoostIO/Reader.cpp` file.

2.7.5 Conclusion

Accordingly, you have now the requirements to serialize data in the framework and a basic knowledge about the mechanism behind it. To conclude, this is a diagram of the serialization mechanism:



2.8 Medical patient folder

DICOM is a software integration standard that is used in Medical Imaging. All modern medical imaging systems (aka Imaging Modalities) equipment like X-Rays, Ultrasounds, CT (Computed Tomography), and MRI (Magnetic Resonance Imaging) support DICOM and use it extensively. The core of DICOM is a file format and a networking protocol.

All Medical Images are saved in DICOM format. Medical Imaging Equipment creates DICOM files. Doctors use DICOM Viewers, computer software applications that can display DICOM images.

> DICOM files contain more than just images. Every DICOM file holds patient information (name, ID, sex and birth date), important acquisition data (e.g., type of equipment used and its settings), and the context of the imaging study that is used to link the image to the medical treatment it was part of.

Roni Z. 2011. Introduction to DICOM [4]:

The objects representing the medical patient data In FW4SPL are aligned with the DICOM standard. In the library `fwMedData` several structures and values have been retrieved:

- `Patient`: name, primary hospital identification number, birth date and sex.
- `Study`: unique identifier of the study, study date and time, referring physician, institution-generated description, age of the patient.
- `Equipment`: institution where the equipment that produced the composite instances is located.
- `Series`: unique identifier of the series, type of equipment that originally acquired the data used to create this series, series date and time, series description, name of the physician(s) administering the series.

In FW4SPL, the class `Series` is the main structure and contains pointers to Patient, Study and Equipment structure. The class `SeriesDB` is a container holding several instances of the `Series` class.

To specify an object of type `Series`, the library `fwMedData` holds the following classes inherited from `Series`:

- `ImageSeries` which corresponds to the image series of DICOM (CT images, MRI images, etc).
- `ModelSeries` which corresponds to the meshes series of DICOM and also represents 3D patient models.

The `fwMedData` library also provides a custom series called `ActivitySeries`. An `ActivitySeries` is a `Series` linked to an activity (sub part an application). Hence it is possible to save the state of all the objects used in the activity. Further application specific parameters which are not referred to an object can also be saved in an `ActivitySeries`. Application parameters in relation to the patient can be the view point on an organ, landmarks, calculated distances between organ points, etc.

2.9 Manager and updater services

2.9.1 Concepts

In the FW4SPL architecture, there is an object container which is often used: `::fwData::Composite`. This container is also an Object and represents a map which associates a string with an Object. The architecture provides two main services to manage a Composite: a composite updater and a service manager.

Updater

The updater service extends service type `::ctrlSelection::IUpdaterSrv` and the work on a selection composite. This kind of service listens specific events from objects identified by their UID. When it receives an event, it performs an operation on an object in the selection composite and notifies composite listeners.

Available operations on composite are:

- Adding an object

[4] Roni Z. 2011. Introduction to DICOM. Introduction. http://dicomiseasy.blogspot.fr/2011/10/introduction-to-dicom-chapter-1.html

- Swapping an object
- Removing an object
- Removing an object if present
- Adding or swapping an object
- Doing nothing

There are few generic updater services which listen all events sent by Objects, and few other which work with particular Object events.

Manager

The manager services extend service type `::ctrlSelection::IManagerSrv` and react to updater messages. This kind of service manages services on identified data if they are present in a composite. There are few manager services, but the most common is `::ctrlSelection::manager::SwapperSrv`. This service manages other services on objects stored in the composite. When this manager gets notified, it can perform an action defined in the manager configuration on the concerned object such as :

- starting the services of the concerned object
- stopping the services of the concerned object
- create communication connection between new objects and/or new services

2.9.2 Implementation

Updater

An updater implementation must inherit from the `::ctrlSelection::IUpdaterSrv` service.

In the example below, an updater is used to manage a `::fwData::Reconstruction` object identified with the `reconstruction` key in a selection composite. This `::fwData::Reconstruction` is stored in a `::fwMedData::ModelSeries` and we used a specific updater to listen events and manage the structure.

It defines two scenarios, each of them belonging to the `<update>` XML tag:

- when the updater receives a `NEW_RECONSTRUCTION_SELECTED` event from the `::fwMedData::ModelSeries` object with uid `model_uid`, it adds or swaps the `rec` object of the selection composite with the object from which it received the event.
- when the updater receives a `REMOVED_RECONSTRUCTIONS` event from the `::fwMedData::ModelSeries` object with uid `model_uid`, it removes the `rec` object of the selection composite if it is present.

Updater configuration example:

```
<object id="model_uid" type="::fwMedData::ModelSeries" />

<object type="::fwData::Composite">

  <service uid="updater_uid"
    impl="::ctrlSelection::updater::SReconstructionFromModelSeriesUpdater"
    type="::ctrlSelection::IUpdaterSrv">
    <update compositeKey="rec"
        onEvent="NEW_RECONSTRUCTION_SELECTED"
        fromUID="model_uid"
        actionType="ADD_OR_SWAP"
```

```
        />
    <update compositeKey="rec"
        onEvent="REMOVED_RECONSTRUCTIONS"
        fromUID="model_uid"
        actionType="REMOVE_IF_PRESENT"
        />
  </service>

</object>

<!-- connect updater to listen the model series -->
<connect>
    <signal>model_uid/objectModified</signal>
    <slot>updater_uid/receive</slot>
</connect>
```

Manager

Managers inherit from `::ctrlSelection::IManagerSrv`. As explained earlier, they manage tasks or services on objects which appear or disappear from the composite on which they are working. For instance, the XML configuration below manages a GUI to configure rendering options of a reconstruction from a reconstruction list thanks to the `::ctrlSelection::manager::SwapperSrv` service.

In this configuration, the manager updates the services attached to the `rec` object each time it is added, removed or swapped.

Manager configuration example

```
<object type="::fwData::Composite">
  <service uid="manager_uid" impl="::ctrlSelection::manager::SwapperSrv"
        type="::ctrlSelection::IManagerSrv"
        autoConnect="yes" >
        <mode type="dummy" />
        <config>
            <object id="rec" type="::fwData::Reconstruction">
                <service uid="organMaterialEditor"
                    impl="::uiReconstruction::OrganMaterialEditor" />
                <service uid="representationEditor"
                    impl="::uiReconstruction::RepresentationEditor" />
            </object>
    </config>
  </service>
</object>
```

Note: Manager mode is *dummy* (`<mode type="dummy">`). With this configuration, if the `::fwData::Reconstruction` object is not present in the selection composite when the manager starts, it will instantiate a new one. In *stop* mode, the manager starts services when the object is present in the selection composite. In *startAndUpdate* mode, the manager exhibits the same behavior as in *stop* mode but also updates services.

2.10 Component-based software

The FW4SPL is also a component-based architecture.

Component-based software engineering (CBSE) (also known as component-based development (CBD)) is a branch of software engineering that emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software. Excerpt from "Component-based software engineering" [5] on Wikipedia

2.10.1 Definitions and characteristics

An individual software component is a software package that encapsulates a set of related code: resources, objects, services, XML configuration, etc.

All the architecture is placed into separate components so that all of the data and functions inside each component are semantically related. Because of this principle, it is often said that components are modular and cohesive.

Components communicate with each other via interfaces. When a component offers services to the rest of the system, it adopts a provided interface which specifies services that other components can use. The generic architecture provided by classes Object/IService and the factory system make this interfacing easier.

Re-usability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them.

2.10.2 Component-based implementation

Implementation requires a dynamic structure which represents the component and a software launcher which loads and manages these components. A component, called a bundle, is just a simple folder that contains :

- the component description file (plugin.xml) to describe the content of the dynamic library
- the dynamic library, the type of which (.so, .dll, .dylib) differs between operating systems
- other shared resources (icons, XSD file, media files, ...)

The software launcher uses the library `fwRuntime` to parse the software description file (profile.xml) and load required dynamic libraries:

```
./launcher.exe mySoftware/profile.xml
```

The component description file (plugin.xml) is used to describe the content of the dynamic library. This file reveals which concepts and concept implementations are proposed by the component. These terms are identified in the file by keywords:

- Extension point: the concept
- Extension: a concept implementation (there can be many implementations one of a single concept)

In some cases, the Extension point is represented by an abstract class in a component, and the Extension by the class that it inherits from the abstract class of another component.

One example is the service concept. The component description file of servicesReg introduces the concept of service and incorporates the class IService into the dynamic library:

```
<plugin id="serviceReg">

  <library name="servicesReg" />

  <extension-point id="::fwServices::registry::ServiceFactory" />
```

[5] Component-based software engineering http://en.wikipedia.org/wiki/Component-based_software_engineering

```
</plugin>
```

And in another component, a new service is proposed in the dynamic library and the information is shared in the description file.

```
<plugin id="myBundle">

   <library name ="myBundle" />

   <! -- myBundle requires the bundle servicesReg to run -->
   <requirement id="servicesReg" />

   <! -- Need code related to ::io::IReader -->
   <requirement id="io" />

   <extension implements =" ::fwServices::registry::ServiceFactory ">
       <! -- service type -->
       <type>::io::IReader</type>
       <! -- the service name available in this component library -->
       <service>::myBundle::myReader</service>
       <! -- the object type associated to the service -->
       <object>::fwData::myData</object>
       <desc>Description of my reader</desc>
   </extension>

</plugin>
```

Even if it is often the case, concepts are not limited to class level. A lot a concepts can be defined : service configurations, operator parameters, etc.

2.11 Graphical User Interface

2.11.1 Overview

Graphical User Interface (GUI) is the process of displaying the graphical components of an application. In fw4spl, the `fwGui` library provides abstract tools to display components like windows, buttons, textfield, aso.

The software architecture provides a way of selecting different backends in order to manage the GUI components. As a result, the `fwGuiQt` library has been created to display components created using the Qt soup. Presently, this backend is the only one supported by the applications.

2.11.2 Backend

When creating an application, we need to specify which gui backend we want to use. To do so, the chosen gui bundle must be activated and started in the profile.xml of the application. The main gui bundle for any application is `guiQt`. The `gui` bundle must be activated regardless of the chosen backend.

```
<activate id="gui" version="0-1" />
<activate id="guiQt" version="0-1" />

<!-- ... -->

<start id="guiQt" />
```

Warning : The gui backend bundle must be started before any other bundle in the profile.xml.

2.11.3 Configuration

Frames

The frame is the main component of a GUI. The main service used to represent a general frame is `::fwGui::IFrameSrv`. The service `::gui::frame::DefaultFrame` is the default implementation for the main application frame. Every backend must provide its own implementation of this service.

The DefaultFrame service is configurable with different parameters :

- Application name
- Application icon
- Minimum window size
- GUI elements (toolbar, menubar, aso.)

```
<service uid="mainFrame" type="::fwGui::IFrameSrv"
    impl="::gui::frame::DefaultFrame" autoConnect="no" >
    <gui>
        <frame>
            <name>Application name</name>
            <icon>path_to_application_icon</icon>
            <minSize width="800" height="600"/>
        </frame>
        <menuBar />
        <toolBar >
            <toolBitmapSize height= "32" width="32" />
        </toolBar>
    </gui>
    <registry>
        <menuBar sid="menuBar" start="yes" />
        <toolBar sid="toolBar" start="yes" />
        <view sid="view" start="yes" />
    </registry>
</service>
```

Menus and actions

The menu bar is used to organize application action groups. The main service used to display that kind of bar is `::fwGui::IMenuBarSrv`. The service `::gui::aspect::DefaultMenuBarSrv` is the default implementation. Every backend must provide its own implementation of this service.

The configuration is used to associate a menu label with the service representing the menu.

```
<service uid="menuBar" type="::fwGui::IMenuBarSrv"
    impl="::gui::aspect::DefaultMenuBarSrv" autoConnect="no" >
    <gui>
        <layout>
            <menu name="First Menu"/>
            <menu name="Second Menu"/>
        </layout>
    </gui>
    <registry>
        <menu sid="firstMenu" start="yes" />
        <menu sid="secondMenu" start="yes" />
    </registry>
</service>
```

The main service used to display a menu is `::fwGui::IMenuSrv`. The service `::gui::aspect::DefaultMenuSrv` is the default implementation. Every backend must provide its own implementation of this service.

The configuration is used to associate an action name and the service performing the action. An action can be configured with a shortcut, a style (default, check, radio) and/or an icon. Several special actions can also be specified (QUIT, ABOUT, aso.).

```
<service uid="myMenu" type="::fwGui::IMenuSrv"
    impl="::gui::aspect::DefaultMenuSrv" autoConnect="no" >
    <gui>
        <layout>
            <menuItem name="First Item" icon="icon_path" />
            <menuItem name="Checked Item" style="check" />
            <separator />
            <menuItem name="Quit" shortcut="Ctrl+Q" specialAction="QUIT" />
        </layout>
    </gui>
    <registry>
        <menuItem sid="actionFirstItem" start="no" />
        <menuItem sid="actionCheckedItem" start="no" />
        <menuItem sid="actionQuit" start="no" />
    </registry>
</service>
```

A menu can also be displayed using a tool bar. The main service used to display a tool bar is `::fwGui::IToolBarSrv`. The service `::gui::aspect::DefaultToolBarSrv` is the default implementation. Every backend must provide its own implementation of this service.

The configuration of a tool bar is the same as the one used to describe a menu.

Layouts

The layouts are used to organize the different parts of a GUI. The main service used to manage layouts is `::fwGui::IGuiContainerSrv`. The service `::gui::view::DefaultView` is the default implementation. Every backend must provide its own implementation of this service.

Several types of layout can be used :

- Line layout
- Cardinal layout
- Tab layout

Every layout can be configured with a set of parameters (orientation, alignment, aso.).

```
<service uid="subView" type="::gui::view::IView"
    impl="::gui::view::DefaultView" autoConnect="no" >
    <gui>
        <layout type="::fwGui::LineLayoutManager" >
            <orientation value="horizontal" />
            <view caption="view1" />
            <view caption="view2" />
        </layout>
    </gui>
    <registry>
        <view sid="subView1" start="yes" />
        <view sid="subView2" start="yes" />
```

```
    </registry>
</service>
```

2.11.4 Multi-threading

The `fwGui` library has been designed to support multi-thread application. When a GUI component needs to be accessed, the function call must be encapsulated in a lambda declaration as shown in this example:

```
::fwGui::registry::Worker::get()->postTask<void>(
[&] {
        //TODO Write function calls
}
).wait();
```

This encapsulation is required because all access to GUI components must be performed in the thread containing the GUI. It moves the function calls from the current thread, to the GUI thread.

CHAPTER 3

Coding style

3.1 Terminology

- Rules are mandatory. Any rule can be (exceptionally) exceeded, but if so, it has to be rigorously justified.
- Recommendations are optional.
- **Camel case** is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter. In programming languages, **camel case** is assumed to start with a lowercase letter. We will use the term **upper camel case** when it starts with a capital.

```
camelCaseLabel
UpperCamelCaseLabel
```

3.2 Generalities

Rule 44 [Preferred language] English is the preferred language for types, variables, functions naming, and code comments.

Rule 45 [Maximum size of a line] A source code line must not exceed 120 characters.

Rule 46 [Indentation] Use only spaces, and an indent level has four spaces.

3.3 C++ coding

3.3.1 Source and files

Rule 4 [Files tree] Source files must be placed in a folder `src/`. Public header files must be placed in a folder `include/`. Private headers may be placed in a different location.

Rule 5 [Files hierarchy] The file hierarchy should follow the namespace hierarchy. For instance, the implementation of a class `::ns1::ns2::SService` should be put in `src/ns1/ns2/SService.cpp`.

Rule 6 [Files extensions] Header files use the extension `.hpp`.

Implementation files use the extension `.cpp`.

Files containing implementation of "template" classes use the extension `.hxx`.

Recommendation 2 [Only one class per file] It is recommended to declare (or to implement) only one class per file. However tiny classes may be declared inside the same file.

Rule 7 [Includes] Use the right include directive depending on the context. `#include "..."` must be used to import headers from the same module, whereas `#include <...>` must be used to import headers from other modules.

Rule 8 [Include path] The include path is not an absolute path depending on a local file system. A correct include path does respect the letter case of the filenames and folders (since some platforms require it) and uses the character '/' as a separator.

Rule 9 [Protection against multiple inclusions] You must protect your files against multiple inclusions. To this end, use the standard directives of the precompiler `#ifndef` and `#define` (since `#pragma` once is only supported by Microsoft compilers).

Use the name of the file and the namespace hierarchy inside the define name in order to prevent any conflict with a file which has the same name but located in a different namespace. Namespaces and file name must be separated by a single underscore _. The define name must be prefixed and suffixed by two underscores __. Last, a comment must be placed after `#endif` to quote the define.

```
#ifndef __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__ // Preamble protecting against
#define __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__ // multiple inclusions.

#endif // __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__
```

Recommendation 3 [Independent headers] A header should compile alone. All necessary includes should be contained inside the header itself. In the following sample :

```
// Header.hpp

class Foo
{
public:
    std::string m_string;
}
```

you will be forced to include the file in this way to get a successful build :

```
// Source.hpp

#include <string>
#include "Header.hpp"
```

This is a bad practice, the header should rather be written :

```
// Header.hpp

#include <string>

// Header.hpp
class Foo
{
public:
    std::string m_string;
}
```

So that people can simply include the header :

```
// Source.hpp

#include "Header.hpp"
```

Recommendation 4 [Minimize inclusions] Try to minimize as much as possible inclusions inside a header file. Include only what you use. Use *forward declarations* when you can (i.e. a type or class structure is not referenced inside the header). This will limit dependency between files and reduce compile time. Hiding the implementation can also help to minimize inclusions (see *Hide implementation*)

Rule 10 [Sort headers inclusions] You must sort headers in the following order : same module, framework libraries, bundles, external libraries, standard library. This way, this helps to make each header independent. The rule can be broken if a different include order is necessary to get a successful build.

```
#include "currentModule.hpp"

#include <libSampleB/second.hpp>
#include <libSampleA/first.hpp>
#include <libSampleB/subModule/first.hpp>

#include <Qt/QtGui>
#include <vector>
#include <map>
```

Recommendation 5 [Sort inclusions alphanumerically] In addition to the previous sort, you may sort includes in alphanumerical order, according to the whole path. Thus they will be grouped by module. For a better readability, an empty line can be added between each module.

```
#include "currentModule.hpp"

#include <libSampleA/first.hpp>
#include <libSampleB/second.hpp>

#include <libSampleB/subModule/first.hpp>
#include <libSampleB/subModule/second.hpp>

#include <Qt/QtGui>

#include <map>
#include <vector>
```

3.3.2 Naming conventions

Rule 11 [Class] Class names must be written in upper camel case. It should not repeat a namespace name. For instance `::editor::SCustomEditor` should be rather called `::editor::SCustom`.

Rule 12 [File] The name of the file should be based on the class name defined in it. It must follow the same letter case.

Rule 13 [Namespace] Namespaces must be written in camel case. A comment quoting the namespace must be placed next to the ending '}'.

```
namespace namespaceA
{
namespace namespaceB
{
    class Sample
    {
    ...
    };
} // namespace namespaceB
} // namespace namespaceA
```

When referring a namespace, you must put `::` if this is a root namespace, with an exception for `std` namespace. Ex: `::boost::filesystem`.

Rule 14 [Function and method names] Functions and methods names must be written in camel case.

Recommendation 6 [Correct naming of functions] Try as much as possible to help the users of your code by using comprehensive names. You may for instance help them to indicate the cost of a function. A function that executes a search to retrieve an object must not be called like a getter. In this case, it is better to call it `findObjet()` instead of `getObject()`.

Rule 15 [Variable] Variable names must be written in camel case. Members of a class are prefixed with a `m_`.

```
class SampleClass
{
private:
   int m_identifier;
   float m_value;
};
```

Static variables are prefixed with a `s_`.

```
static int s_staticVar;
```

Rule 16 [Constant] Constant variables must be written in snake_case but in capitals, and follow the previous rule.

```
class SampleClass
{
    static const int s_AAA_BBB_CCC_VALUE = 1;
};

void fooFunction()
{
    const int AAA_BBB_VAR = 1;
    ...
}
```

Rule 17 [Type] Type names, like classes, must be written in upper camel case.

```
typedef int CustomType;
typedef vector<int> CustomContainer;
```

Rule 18 [Template parameter] Template parameters must be written in capitals. In addition, they must be short and explicit.

```
template< class KEY, class VALUE > class SampleClass
{
    ...
};
```

Rule 19 [Macro] Macros without parameters must be written in capitals. On the contrary, there is no specific rule on macros with parameters.

```
#define CUSTOM_FLAG_A 1
#define CUSTOM_FLAG_B 1

#define CUSTOM_MACRO_A( x ) x
#define Custom_Macro_B( x ) x
#define custom_Macro_C( x ) x
#define custom_macro_d( x ) x
```

Rule 20 [Enumerated type] An enumerated type name must be written in upper camel case. Labels must be written in capitals. If a `typedef` is defined, it follows the upper camel case standard.

```
typedef enum SampleEnum
{
    LABEL_1,
    LABEL_2
    ...
} SampleEnumType;
```

Rule 21 [Service] A service implementation is identified by a `S` at the beginning of the class name. Example : `SCustomEditor`. A service interface is identified by a `I` at the beginning of the class name. Example : `IEditor`.

Rule 22 [Signal] A signal name must be prefixed with `sig`. It should be suffixed by a past action (ex: Updated, Triggered, Cancelled, CakeCookedAndBaked). It follows other common variable naming rules (member of a class, etc...).

```
class Sample
{
    SigType::sptr m_sigImageDisplayed;
};
```

Rule 23 [Slot] A slot name must be prefixed with `slot`. It should be suffixed by an imperative order (Ex: Update, Run, Detach, Deliver, OpenWebBrowser, GoToFail). It follows other common variable naming rules (member of a class, etc...).

```
class Sample
{
    SlotType::sptr m_slotDisplayImage;
}
```

3.3.3 Coding rules

Blocks

Rule 24 [Indentation] Code block indentation and bracket positioning follow the Allman style.

```
void function(void)
{
    if(x == y)
    {
        something1();
        something2();
    }
    else
    {
        somethingElse1();
        somethingElse2();
    }
    finalThing();
}
```

Rule 25 [Indentation of namespaces] Namespaces are an exception of the previous rule. They should not be indented.

```
namespace namespaceA
{
namespace namespaceB
```

```
{
    ...
} // namespace namespaceB
} // namespace namespaceA
```

Rule 26 [Blocks are mandatory] After a control statement (if, else, for, while/do...while, try/catch, switch, foreach, etc...), it is mandatory to open a block, whatever is the number of instructions inside the block.

Rule 27 [Scope] The keywords `public`, `protected` and `private` are not indented, they should be aligned with the keyword `class`.

```
class Sample
{
public:
    ...
private:
    ...
};
```

Class declaration

Recommendation 7 [Only three scope sections] When possible, use only one section of each scope type `public`, `protected` and `private`. They must be declared in this order.

Recommendation 8 [Group class members by type] You may group class members in each scope according to their type: type definitions, constructors, destructor, operators, variables, functions.

Rule 28 [Hide implementation] Avoid non-const public member variables except in very small classes (i.e. a 3D point). The Pimpl idiom may also be helpful to separate the implementation from the declaration.

Recommendation 9 [Hide implementation] Try to put variables as much as possible in the `private` section.

Rule 29 [Accessors] Since you protect your member variables from the outside, you will have to write accessors, named `getXXX()` and `setXXX()`. Getters are always `const`.

Rule 30 [Template class function definition] The function definition of a template class must be defined after the declaration of the class.

```
template < typename TYPE >
class Sample
{
public:
    void function(int i);
};

template < typename TYPE >
inline Sample<TYPE>::function(int i)
{
    ...
}
```

Recommendation 10 [Separate template class function definition] In addition of the previous rule, you may put the definition of the function in a `.hxx` file. This file will be included in the implementation file right after the header file (the compile time will be reduced comparing with an inclusion of the `.hxx` in the header file itself).

```
#include <namespaceA/file.hpp>
#include <namespaceA/file.hxx>
```

Initializer list

Rule 31 [One initializer per line] In a class constructor, use the initialization list as much as possible. Place one initializer per line. Constructors of base classes should be placed first, followed by member variables. Do not specify an initializer if it is the default one (empty std::string for instance).

```
SampleClass::SampleClass( const std::string& name, const int value ) :
    BaseClassOne( name ),
    BaseClassTwo( name ),
    m_value( value ),
    m_misc( 10 )
{}
```

Recommendation 11 [Align everything that improves readability] To improve readability, you may align members on one hand and argument lists on the other hand.

```
SampleClass::SampleClass( const std::string& name, const int value ) :
    BaseClassOne  ( name ),
    BaseClassTwo  ( name ),
    m_value       ( value ),
    m_misc        ( 10 )
{}
```

Functions

Rule 32 [Constant reference] Whenever possible, use constant references to pass arguments of non-primitive types. This avoids useless and expensive copies.

```
void badFunction( std::vector<int> array )
{
    ...
}

void goodFunction( const std::vector<int>& array )
{
    ...
}
```

Recommendation 12 [Constant reference for shared pointers] For performance sake, it is preferable to use `const&` to pass arguments of type `::boost::shared_ptr`. It is only useful to pass the pointer by copy if the pointer can be invalidated by an another thread during the function call. If you have any doubt, it is safer to pass the argument by copy.

Rule 33 [Constant functions] Whenever a member function should not modify an attribute of a class, it must be declared as `const`.

```
void readOnlyFunction( const std::vector<int>& array ) const
{
    ...
}
```

Recommendation 13 [Limit use of expression in arguments] When passing arguments, try to limit the use of expressions to the minimum.

```
// This is bad
function( fn1(val1 + val2 / 4 ), fn2( fn3( val3 ), val4) );

// This is better
```

```
const float res0 = val1 + val2 / 4;

const float res1 = fn1(res0);
const float res3 = fn3(val3);
const float res2 = fn2(res3, val4);

function( res1 , res2 );
```

Miscellaneous

Rule 34 [Enumerator labels] Each label must be placed on a single line, followed by a comma. If you assign values to labels, align values on the same column.

```
enum OpenFlag
{
    OPEN_SHARE_READ      = 1,
    OPEN_SHARE_WRITE     = 2,
    OPEN_EXISTING        = 4,
};
```

Rule 35 [Use of namespaces] You have to organize your code inside namespaces. By default, you will have at least one namespace for your module (application or bundle). Inside this namespace, it is recommended to split your code into sub-namespaces. This helps notably to prevent naming conflicts.

It is forbidden to use the expression‘‘using namespace‘‘ in header files but it is allowed in implementation files. It is however recommended to use aliases in this latter case.

```
namespace bf = ::boost::filesystem;
```

Rule 36 [Keyword const] Use this keyword as much as possible for variables, parameters and functions.

Recommendation 14 [Keyword auto] Use this keyword as much as possible to improve maintainability and robustness of the code.

Rule 37 [Prefer constants instead of #define] Use a static constant object or an enumeration instead of a `#define`. This will help the compiler to make type checking. You will also be able to check the content of the constants while debugging. You can also define a scope for them, inside the namespace, inside a class, private to a class, etc...

Rule 38 [Prefer references over pointers] When possible, use references instead of pointers, especially for function parameters. Pointer as parameter should only be used if it is considered to have a NULL pointer or when passing a C-like array. If you use a pointer, always check it if is null in the current scope before dereferencing it.

Rule 39 [Type conversion] For type conversion, use the C++ operators which are `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. Use them wisely in the appropriate case. You may read this documentation.

Recommendation 15 [Strings to numbers/numbers to string conversion] When converting strings to numbers or numbers to string, prefer the use of boost::lexical_cast.

Recommendation 16 [Exceptions] Exceptions are the preferred mechanism to handle error notifications.

Rule 40 [Explicit integer types] When you do need a specific integer size, use type definitions declared in <cstdint>, for example :

Bits	Signed	Unsigned
8	int8_t	uint8_t
16	int16_t	uint16_t
32	int32_t	uint32_t
64	int64_t	uint64_t

3.4 Documentation

Rule 41 [Document the code] The code must be documented with **Doxygen**, an automated tool to generate documentation.

Rule 42 [Location of the documentation] Every documentation that can be useful to a user must be placed inside the header files. Thus a user of a module can find the declaration of a class and its documentation at the same place. Inside the implementation file, the documentation will give more details about algorithms. Moreover, every documentation must be placed next to the entity it is refering to, in order to help searching inside the code.

Recommendation 17 [Lightweight documentation] Inside a documentation block, only use necessary tags. This will avoid to overload the documentation and makes it readable. By the way, empty tags will be presented inside the generated documentation and will be useless. Just use an empty line to make a separation inside a documentation block. Don't indicate parameter types when using `@param` directive. This is useless since it will duplicate information of the function prototype. Also, prefer the use of `///` whenever possible.

Example 1 : Bad documentation block

```
/**
* @brief        A very short description.
*
* A longer description, giving more details about the documented piece
* of code.
*********************************************
* @param
*********************************************
* @return
*********************************************
* @exception
*********************************************
* @todo
*********************************************
```

Example 2 : Good documentation block

```
/**
* @brief        A very short description.
*
* A longer description, giving more details about the documented piece
* of code.
*/
```

Example 3 : Function documentation

```
class Sample
{
public:
    /**
    * Retrieve the thing.
    *
    * @return       The thing value.
```

```
        */
        const std::string& getThing( void ) const;
        /**
        * @brief        Set the thing.
        *
        * @param        thing   :  The new thing.
        */
        void setThing( const std::string& thing );

    private:
        /// stored thing
        std::string     m_thing;
    };
```

Recommendation 18 [Structured documentation] Doxygen provides a default structure when you generate the documentation. However, when dealing with a big documented entity, it is often recommended to use the group feature (`@name`). With this feature you will build a logical view of the class interfaces.

Rule 43 [Document service configuration] The method `configuring` of a service must be properly documented. It should indicate every parameter that can be passed, no matter if it is optional or not. Example :

```
    /**
     * @verbatim
    <adaptor id="points" class="::namespace::SService">
        <config option1="default" option2="false"/>
    </adaptor>
     @endverbatim
     * - \b option1 : first option.
     * - \b option2(optional) : second option.
    */
    NAMESPACE_API void configuring() throw(fwTools::Failed);
```

3.5 XML coding

Rule 48 [Id name] Id should have a semantic name. Avoid id like myXXXXX or customXXXXX. Moreover, id must be written in lower case with an underscore as separator.

```
    <service id="generic_scene" />
```

3.6 CMakeLists coding

Rule 1 [Function name] Standard CMake functions and macros should be written in lower case. Each word is generally separated by an underscore (this is a rule of CMake anyway).

```
    add_subdirectory("library/")
    include_directories(SYSTEM "/usr/local")
```

Rule 2 [Macro name] Custom macros should be written in camel case.

```
    fwLoadProperties()
    fwLink("boost")
```

Rule 3 [Variable name] Variables should be written in upper case letters separated if needed by underscores.

```
set(VARIABLE_NAME "")
```

Recommendation 1 [Expression in block ending] In the past, CMake enforced to specify the label or expression in block ending, for instance :

```
function(name arg1 arg2)
    ...
    if(expr1)
        ...
    else(expr1)
        ...
    endif(expr1)
    ...
endfunction(name)
```

This is no longer needed in latest CMake versions, and we recommend to use this possibility for the sake of simplicity.

```
function(name arg1 arg2)
    ...
    if(expr1)
        ...
    else()
        ...
    endif()
    ...
endfunction()
```

3.7 Licence

Rule 47 [LGPL] Do not forget to put the LGPL licence block on fw4spl.

```
/* ***** BEGIN LICENSE BLOCK *****
 * FW4SPL - Copyright (C) IRCAD, 2009-2015.
 * Distributed under the terms of the GNU Lesser General Public License (LGPL) as
 * published by the Free Software Foundation.
 * ***** END LICENSE BLOCK ***** */
```

CHAPTER 4

Frequently Asked Questions (FAQ)

4.1 What is fw4spl?

The framework FW4SPL (FrameWork for Software Production) is an open-source framework, developed by IRCAD (research institute against cancer and disease). The principle of FW4SPL is the fast and easy creation of applications, mainly in the medical field. Therefore it provides features like digital image processing in 2D and 3D, visualization or simulation of medical interactions. To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

4.2 What does fw4spl mean?

FW4SPL means FrameWork for Software Production Line. It is also called F4S ("forces").

4.3 What are the features of fw4spl?

The framework is built around the notion of component (bundle). To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

FW4SPL has a component-based architecture composed of C++ libraries. There are three main concepts in the architecture: - object-service concept - component approach - signal-slot communication

4.4 Which platforms does fw4spl run on?

This framework can run under Windows, Linux and MacOS and we are working on the Android part.

4.5 Where can I find applications developed with fw4spl ?

Some tutorials are provided with the framework and you can also build VR-Render, a free visualization software.

4.6 Which prerequiste do I need to develop bundle?

You must have a good knowledge in C++. Concerning the configuration files, they are written in XML.

4.7 What are the BinPkgs?

The BinPkgs (binary packages) contain all the extern libraries needed by fw4spl. For each BinPkg, a CMakeLists provides the OS specific instructions to build it . They can be downloaded on https://github.com/fw4spl-org/fw4spl-deps

4.8 Is it difficult to compile an application with fw4spl?

No, it isn't. You just have to compile all the bundles and libraries used by the application.

4.9 Why does fw4spl provide a launcher?

The launcher is used to create the entry point of the application. It parses the profile and configuration xml file to build it.

4.10 How can I debug my program ?

First, you can change the log level of a sub-project in the CMake configuration.

The allowed values are : ['trace', 'debug', 'info', 'error', 'fatal', 'warning', 'disable']. the value 'trace' gives me the maximun of log, 'disable' disables log.

> **note a** : Printing many log messages (by activating trace on all sub-projects for ex.) can be very time consuming for the application.
>
> > **note b** : Under windows system, log messages are saved on filesystem in SLM.log file, in the working directory.

Secondly, you can debug your application using gdb (Linux/Mac) or Visual Studio (Windows) and compiling your application in Debug mode

> **note a** : you can use gdb like this "LD_LIBRARY_PATH=./lib gdb -arg bin/launcher Bundles/myApp/myProfile.xml", and press "r" for run the program
>
> **note b** : you can use under gdb the command "catch throw" hence gdb will stop near the error **note c** : Documentation to learn using gdb : http://www.cs.tau.ac.il/lin-club/lecture-notes/GDB_Linux_telux.pdf

Thirdly, you can manage the program complexity by reducing the number of activated components (in profile.xml) and the number of created services (in config.xml) to better localize errors.

Fourthly, verify that your profile.xml / plugin.xml and each bundle plugin.xml are well-formed, by using xmllint (command line tool provided by libxml2).

4.11 I have an assertion/fatal message when I launch my program, any idea to correct the problem ?

First, you can read the output message :) and try to solve the problem. In many cases, there are two kind of problems. The program fails to :

- create the service given in configuration In this case, four reasons are possibles :

  - the name of service implementation in config.xml contains mistakes
  - the bundle that contains this service is not activated in the profile
  - the bundle plugin.xml, that contains this service, not declares the service or the declaration contains mistakes.
  - the service is not register in the Service Factory (forget of macro REGISTER_SERVICE(...) in file .cpp)
- manage the configuration of service. In this case, the implementation code in .cpp file (generally configuring() method of service) does not correspond to description code in config.xml (Missing arguments, or not well-formed, or mistakes string parameters).

4.12 If I use fw4spl, do I need wrap all my data ?

The first question is to know if the data is on center of application:

- Need you to shared data between few bundles ?
- Need you to attach services on this data ?
  - If the answer is no, you don't need to wrap your data.
  - Otherwise, you need to have an object that inherits of ::fwData::Object.

In this last case, do you need shared this object between different services which use different libraries, ex for Object Image : itk::Image vs vtkImage ?

- If the answer is yes, you need create a new object like fwData::Image and a wrapping with fwData::Image<=>itk::Image and fwData::Image<=>vtkImage.
- Otherwise, you can just encapsulated an itk::Image in fwData::Image and create an accessor on it. (however, this kind of choice implies that all applications that use fwData::Image need itk library for running.)

CHAPTER 5

How to use CMake with Fw4spl

5.1 CMake for fw4spl

5.1.1 Introduction

Fw4spl and it's dependencies are based on CMake . Note that the minimal version of cmake to have is 2.8.12.

5.1.2 CMake files for dependencies

fw4spl dependencies are based on the ExternalProject concept from lastest versions of cmake.

The concept is to create custom targets to build projects in external trees. Each project has custom steps for download, update/patch, configure, build and install.

Here is a simple example from camp :

```
cmake_minimum_required(VERSION 2.8)

project(campBuilder)

include(ExternalProject)

set(CAMP_CMAKE_ARGS ${COMMON_CMAKE_ARGS}
                    -DBUILD_DOXYGEN:BOOL=OFF
                    -DBOOST_INCLUDEDIR:PATH=${CMAKE_INSTALL_PREFIX}/include/boost-1_57
)

getCachedUrl(https://github.com/greenjava/camp/archive/0.7.1.1.tar.gz CACHED_URL)

ExternalProject_Add(
    camp
    URL ${CACHED_URL}
    DOWNLOAD_DIR ${ARCHIVE_DIR}
    DEPENDS boost
    INSTALL_DIR ${CMAKE_INSTALL_PREFIX}
    CMAKE_ARGS ${CAMP_CMAKE_ARGS}
    STEP_TARGETS CopyConfigFileToInstall
)

ExternalProject_Add_Step(camp CopyConfigFileToInstall
    COMMAND ${CMAKE_COMMAND} -E copy ${CMAKE_SOURCE_DIR}/cmake/findBinpkgs/FindCAMP.cmake ${CMAKE_INS
    COMMENT "Install configuration file"
```

The important parts are in the *ExternalProject_Add* fonction:

- URL: is the download link of the sources
- DOWNLOAD_DIR: The folder where the sources will be stored (set globaly for all deps)
- DEPENDS: The dependencies of the current library (will be compiled before)
- INSTALL_DIR: The folder in which the library will be installed (set globaly for all deps)
- CMAKE_ARGS: CMake options for library which have a cmake build system
- STEP_TARGETS: Custom command (in this example it will copy a script in the install folder)

Note that in other script you can have much more options like:

- PATCH_COMMAND
- CONFIGURE_COMMAND
- BUILD_COMMAND
- INSTALL_COMMAND

Refer you to the documentation of ExternalProject for more informations.

5.1.3 CMake files for fw4spl

Each project (apps, bundles, libs) have two "CMake" files:

- CMakeLists.txt
- Properties.cmake

The CMakeLists.txt file

The CMakeLists.txt should contain at least the function *fwLoadProperties()* to load the Properties.cmake. But it can also contain others functions usefull to link with external libraries.

Here is an example of CMakeLists.txt from guiQt Bundle :

```
fwLoadProperties()
fwUseForwardInclude(
    fwActivities
    fwGuiQt
    fwRuntime
    fwServices
    fwTools

    gui
)

find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)


fwForwardInclude(
    ${Qt5Core_INCLUDE_DIRS}
    ${Qt5Gui_INCLUDE_DIRS}
    ${Qt5Widgets_INCLUDE_DIRS}
)
```

```
fwLink(
       ${Qt5Core_LIBRARIES}
       ${Qt5Gui_LIBRARIES}
       ${Qt5Widgets_LIBRARIES}
)

set_target_properties(${FWPROJECT_NAME} PROPERTIES AUTOMOC TRUE)
```

The first line *fwLoadProperties()* will load the properties.cmake (see explanation in the next section). The *fwUseForwardInclude(...)* function will add the include directories of each argument to the target.

The next lines are for the link with an external libraries (fw4spl-deps), in this example it is Qt.

The first thing to do is to call *find_package(The_lib COMPONENTS The_component)*.

The use *fwForwardInclude* to add includes directories to the target, and *fwLink* to link the libraries with your target.

You can also add custom properties to your target with *set_target_properties*.

The Properties.cmake file

Properties.cmake should contain informations like name, version, dependencies and requirements of the current target.

Here is an example of Properties.cmake from fwData library:

```
set( NAME fwData )
set( VERSION 0.1 )
set( TYPE LIBRARY )
set( DEPENDENCIES fwCamp fwCom fwCore fwMath fwMemory fwTools )
set( REQUIREMENTS  )
```

- NAME: Name of the target
- VERSION: Version of the target
- TYPE: Type of the target (can be library, bundle or executable)
- DEPENDENCIES: Link the target with the given libraries (see target_link_libraries)
- REQUIREMENTS: Ensure that the depends are build before target (see add_dependencies)

5.2 Tutorials

5.2.1 How can I add a new dependency

You may want to add a new dependency into fw4spl-deps or you may want to add your own folder of dependencies.

Tip: You need to know that the main CMakeLists.txt is in fw4spl-deps, and you can add as many additional folders as you want. Use the *ADDITONNAL_DEPS* option in cmake to set the path of your custom deps.

Add a new deps in fw4spl-deps

Adding a new deps is quite easy, the only things to do is to add a new folder *myNewDeps* and put a CMakeLists.txt file into it. The CMakeLists.txt should contain at least:

- cmake_minimum_required()

- project()
- include(ExternalProject)
- ExternalProject_Add(...)

For example:

```
cmake_minimum_required(VERSION 2.8)

project(myDepsBuilder)

include(ExternalProject)

getCachedUrl(http://myDeps.com/myDeps.zip CACHED_URL)

ExternalProject_Add(
    myDeps
    URL ${CACHED_URL}
    DOWNLOAD_DIR Path/To/Your/Download/dir
    PATCH_COMMAND your_patch_command (optional)
    CONFIGURE_COMMAND your_configure_command (optional)
    BUILD_COMMAND  your_build_command (optional)
    INSTALL_COMMAND your_install_command (optional)
    INSTALL_DIR your_install_dir
    CMAKE_ARGS cmake_arguments
)
```

Add a custom deps repository

You may want to add your own folder of dependencies (as fw4spl-ext-deps or fw4spl-ar-deps). In this case your main need to create a CMakeLists.txt in the root of your folder (myDepsFolder/CMakeLists.txt) in order to list the subdirectories of your deps.

```
cmake_minimum_required(VERSION 2.8)

project(CustomDeps)

list(APPEND SUBDIRECTORIES myDeps1)
list(APPEND SUBDIRECTORIES myDeps2)
...
```

Then when you do a *ccmake* or *cmake-gui* in the build of your deps, you need to add the path to your custom repository in the *ADDITONNAL_DEPS* option. Then cmake will automaticaly parsed your folder.

5.2.2 How can I add a custom bundle in fw4spl

You may want to add a new bundle/lib/app in an existing repository or you may want to add your custom repository to fw4spl.

Tip: You need to know that the main CMakeLists.txt is in fw4spl repository, and you can add as many additional repository as you want. Use the *ADDITIONAL_PROJECTS* option in cmake to add path of your custom folders.

Add a new bundle/lib/app in fw4spl

The only thing to do is to write a CMakeLists.txt and a Properties.cmake (see section Cmake for Fw4spl for more informations).

Add a custom repository to fw4spl

As the main CMakeLists.txt is in fw4spl repository,you need to add the path of your folder in *ADDITIONAL_PROJECTS* option when you launch *ccmake* of *cmake-gui* on the build folder of fw4spl. Then your folder will automaticaly be parsed by cmake.

Note: All your bundle/lib/application need to respect the fw4spl-cmake conventions and have a CMakeLists.txt and a Properties.cmake.

CHAPTER 6

Contributors

6.1 Contributors

From 2004 to 2006, an advanced modular software for patient modeling (see publication page) has been designed and implemented by Guillaume Brocker, Johan Moreau, Jean-Baptiste Fasquel, Vincent Agnus and Nicolas Papier. This represented the basis of the component management system of FW4SPL, essentially conceived by Guillaume Brocker and Johan Moreau. This framework version (v0.1) was used to create 3 software tools in visualization and medical image processing in the Eureka project Odysseus (3DVPM, 3DDVP and MARNS software).

Throughout 2007, Vincent Agnus and Jean-Baptiste Fasquel conceived and implemented the main core mechanisms of this new version of FW4SPL. Jean-Baptiste Fasquel focused on the notion of roles coupled with the component management system, the inter-role communication system, as well as an appropriate XML formalism for the description of both roles embedded into components and description of software. Many basic software tools have been built to validate the architecture (see publication page). Vincent Agnus also focused on role design, and more specifically on data structures, a generic serialization mechanism and a powerful template dispatching technique. During his internship in 2007, Benjamin Gaillard has improved the communication system in FW4SPL. In parallel with the work on the pure FW4SPL system, Johan Moreau got involved in the construction/compilation system and, together with Arnaud Charnoz, in the management of external dependencies and some specific medical data structures. Their work also led to advanced visualization of medical images (free download). Early 2008, the framework was available in version 0.2.

During the period from mid-2008 to mid-2009, some advanced data structures and functionalities have been developed on the basis of the architecture to further evaluate it and make it more robust. A larger development team has been involved, including Emilie Harquel, Julien Waechter and Nicolas Philipps additionally to Vincent Agnus, Jean-Baptiste Fasquel, Johan Moreau and Arnaud Charnoz. Additional efforts have been made by Johan Moreau and Arnaud Charnoz on the management of external dependencies. Nicolas Philipps, Julien Waechter and Johan Moreau also improved the construction environment Sconspiracy, initially opened as an opensource project YAMS++ in 2007. The version 0.3 of the framework had been achieved by early summer 2009.

From mid-2009 to mid-2010, the main work on FW4SPL included: performing generic scenes for visualization (mainly developed by Nicolas Philipps, Julien Waechter and Vincent Agnus), a new communication system (mainly developed by Nicolas Philipps and Arnaud Charnoz), new UI components (mainly developed by Emilie Harquel and Julien Waechter), better log and assert system (by Arnaud Charnoz), new documentation (mainly done by Pascal Monnier, Alexandre Hostettler).

FW4SPL (version 0.4) has been opened late 2009 and was used to create several software in the European project Passport (VR-Render, VR-Render WLE, AR-Surg, VR-Planning and VR-Probe software). In December, we had switched to version 0.5 (with generic scene). The latest stable version is 0.6 (new communication system) and the current branch development is the 0.6.1 branch.

Version 0.7 adds a limited Qt support during summer 2010 (Hocine Chekatt's internship) and limited support for Python, OpenNI and SOFA (these two last parts had been developed by Altran). During 2011, FW4SPL 0.8 adds a Qt

based 2D scene (Ivan MATHIEU's internship), new buffer for meshes and images, new memory dump mechanisms, a new set of applications (Apps/Examples), a new Dicom reader (Jordi ROMERA's internship), new registration functionalities (Marc Schweitzer's internship) an improved image origin management, etc. A new scenegraph design has been developed but not yet integrated (Lo?c Velut's internship).

Multithreading (fwThread), signal/slot (fwCom), dump management and data introspection (fwAtoms) mechanisms have been added during 2012 in version 0.9 (co-working between IRCAD and IHU). A new design to manage data and store data (Julien Weinzorn's internship) has been prototyped.

This version supports msvc2010 and has also been used to evaluate the transition to Android and iOS (Adrien Bensaibi's internship). Altran has added a connector towards the management tool of the MIDAS content developed by Kitware. Finally, a version management mechanism has been developed (fwAtomsPatch) (Cl?ment Troesch's internship) and new data has been created (fwMedData). This version has been used by the Visible Patient company within the framework of their ISO 13485 certification. A new repository has also been created (fw4spl-ext) with the aim of welcoming not yet stabilized functionalities or to host PoC. The CMake construction system is also supported.

Version 0.10.0 provides the notion of timeline to manage temporal data (IHU). The SConspiracy construction system has been removed.

	Core, visualization, image processing, applications and tutorials **Team** [Johan Moreau, Marc Schweitzer, Fr?d?ric CHAMP,] Flavien Bridault-Louchez, Pascal Monnier
	Core, visualization, image processing, applications and tutorials Team : Julien Waechter, Emilie Harquel, Jessica GROMER
	**Proof of concept on Kinect and Sofa integration** • Altran_200609_MAG10_FR.pdf French document p12 • Altitude_17_20100407_FR.pdf French document p26 Proof of concept on MIDAS integration
	Team : Nicolas Philipps, Valentin Martinet, Arnaud Charnoz
	This project has partly funded by the European Commission via PASS • http://www.passport-liver.eu/ • http://www.vph-noe.eu/vph-repository/doc_download/154-passportppt • newsletter july 2010

6.2 Libraries

6.3 FLOSS projects using FW4SPL

Project	Description
skuld-project	Skuld project work on mobile port (iphone, android, meego/maemo, ...) of FW4SPL