INRIA INNOVATION LAB

CERTIVIBE

V1.0

SOFTWARE DESIGN DESCRIPTION

Document Approval			
	Name	Function	Date
Originated by	Charles Garraud	Development team	12/11/2015
Reviewed by	Cédric RIOU	Development team	21/09/2017
Approved by	Benoît Perrin	Project Manager	16/02/2018

HISTORY

Version	Author	Date	Comments
01	CG	12/11/2015	Document Creation
02	CRIO	21/09/2017	Review and update

TABLE OF CONTENTS

1. Document Information

1.1 Document Roadmap

1.1.1 Objectives

The main purpose of the document is to describe the general architecture of OpenViBE software system, a plugin-based software framework dedicated to real-time neuroscience.

1.1.2 Document Overview

The document is divided into subsections:

- $1 Document Information provides information about the purpose and use of the document;
- $2 System Purpose describes the main purpose of the system;
- $3 Functional Overview gives a global overview of the system from a functional perspective;
- $4 Technical Overview gives a global overview of the system from a technical perspective;
- $5 Software System Organization presents the system general organization;

Refer to $1.2.2 Views Listing for the list of descriptive views provided in this document.

1.1.3 Document User Guide

This document can be explored in different ways according to stakeholder expectations:

- Scientists and software developers can browse all technical sections in order to get accustomed to the software;

1.1.4 QA team members can explore Document Information section to inspect the scope of the document or browse the

Views are given a unique incremental identifier to be used as reference in other QA documents.

ID	View Reference	Description
SDD-XXX		*overview*
…		

- section to check requirements are handled at architectural level. They might also be interested in the **Erreur ! Source du renvoi introuvable.** section.

Different references can be found in the document:

- SRS-XXX identifiers are used to reference requirements (see SRS);
- RSK-XXX identifiers are used to reference risks;
- SDD-XXX identifiers are used to reference architectural views presented in this document;
- MSD-XXX identifiers are used to reference architectural views presented in the MSD document.

1.1.5 References

DOCUMENT #	TITLE
EN ISO 13485	Quality systems – Medical devices – System requirements for regulatory purposes
EN ISO 14971	Medical Devices - Application of Risk Management to Medical Devices
EN ISO 62304	Medical Device Software - Software Life Cycle Processes
SRS	Software Requirement Specifications
MSD	Module Software Detailed Specifications

1.1.6 Definitions

View	A view is a specific snapshot of the architecture. It describes whole or part of the system from a given perspective.
Prototype	In the set of technical documents, refers to the black-box description of a technical entity (Input/Output description only).
System/Framework/Software system	OpenViBE Software System.

1.1.7 Abbreviations

ASR	Architecture Significant Requirement. It involves requirements in SRS that have a high-impact on the architecture (usually most of the non-functional requirements).
OO	Object Oriented (relates to object oriented programming paradigm).

API	Application Programming Interface.
I/O	Input/Output.

1.1.8 Tools

UML diagrams in this document were generated with plantUML that generates diagram from simple text files.

1.2 Architectural View Documentation

1.2.1 View Description

The system software architecture is documented through snapshots of the system from a given perspective. These snapshots are called views.

The following table presents a short overview of the different type of views:

View Type	**Description**	**Identification**
Logical View	Describes the system statically. It includes system decomposition, layers description, packages and classes hierarchy, black-box module prototype with I/O description.	LV-DiagramType-Identifier *Ex: LV-ClassDiagram-Module1*
Behavioral View	Describes the system dynamically. It includes description of scenarios, communication between entities and state changes.	BV-DiagramType-Identifier *Ex: BV-SequenceDiagram-UseCase1*
Physical View	Describes the system in term of physical entity. It includes all physical entities that map some structural components (e.g. library, executable, source files etc.).	PV-DiagramType-Identifier *Ex:* *PV-ComponentDiagram-Module1* *PV-FileHierarchy-Project*
Implementation Views	Describes the system from an implementation perspective. It includes code snapshots, file format and protocol description.	IV-DiagramType-Identifier *Ex:* *IV-FileFormat-XmlScenario* *IV-SourceCode-MixinPattern*

A view template is available at $6.1 Architectural View Template.

1.2.2 Views Listing

Views are given a unique incremental identifier to be used as reference in other QA documents.

ID	View Reference	Description
SDD-XXX		*overview*
…		

2. System Purpose

OpenViBE is a software system that aims to be used by private companies or research centers with needs in EEG data processing.

EEG data processing is a wide area and it is not possible to forecast what will be these needs. It can vary a lot depending on the final application field (Neurofeedback, P300 etc.).

To overcome this variability problem, OpenViBE offers flexibility with services to:

- Create **personalized** chains of signal processing;
- Use theses personalized chains to process EEG data.

Besides, it is impossible to predict what type of applications OpenViBE users want to develop. For instance, it can be GUI application intended for medical use or a domain specific EEG data processing engine.

Therefore, OpenViBE provides its services through a non-GUI framework distributed as a set of libraries manipulated through C++ APIs. Creating and using processing chains has to be done programmatically which is less intuitive but less restrictive as well.

3. Functional Overview

As described in $2 System Purpose, the software system offers services to create personalized chains of processing and use theses chains to process EEG data. A chains of processing is called pipeline in computing. Here is a short description of this concept:

In computing, a pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one.

Wikipedia - Pipeline (Computing)

3.1 List of features

First, a pipeline is a *set of data processing elements*. Each element is dedicated to a specific purpose (e.g. data acquisition, filtering, writing data etc.). Therefore, the functional capacity of a pipeline-based framework depends on:

- The list of available elements;
- The inter-element compatibility (ability to connect two elements).

In OpenViBE, a pipeline is called **scenario** and a pipeline building block is called a **box.** A **box algorithm** is the software component each box relies on (i.e. the processing engine). Low-level processing unit are called algorithms.

The system provides a plugin mechanism to offer the capacity to extend the functionalities without the need to recompile. Each plugin is a software component that adds specific feature to the system. A plugin can contain the following features:

- Algorithms definitions;
- Box algorithms definitions.

Terminology summary:

Scenario	• Processing pipeline represented as chains of boxes
Box	• Building block of a scenario • Relies on box algorithm
Box algorithm	• Box processing engine
Algorithm	• Low-level unit of processing • Provides a specific service • Can be manipulated directly
Metabox	• *Assembly of boxes* • *Behaves just like a normal box* • *Has an arbitrary number of inputs and outputs using the same types as other boxes* • *Can have an arbitrary number of settings* • *Can be inserted into a scenario*

3.2 Scenario creation

As described in $3.1 List of features, the framework provides a list of features that can be extended. The definition of a pipeline says the elements are *connected in series, where the output of one element is the input of the next one.*

Building a pipeline consists of selecting a subset of features, arranging them and connecting them.

OpenViBE provides this service through a **centralized management module** called Kernel. Two keywords in the Kernel definition allows a better understanding of the whole framework architecture:

- *Management:* The kernel provides some management services through dedicated submodules built around a central manager. It is the brain of the system.
- *Centralized:* There is no direct communication between boxes.

Kernel managers are accessed through a Kernel context.

The manager dedicated to scenario creation is the scenario manager which is part of the scenario management submodule. It provides a convenient API to create a scenario, add boxes and link them. The scenario manager allows scenario to be stored/loaded on the disk.

Typically a scenario needs input data to process. Following input data can be used to feed a scenario:

- Data acquired from an EEG device;
- Data read from a file;
- Data generated on the fly.

3.3 Scenario Use

A scenario is obviously meant to be used at some point. The previous section introduces the Kernel and its managers. The manager dedicated to playing scenarios is the player manager which is part of the scenario playback submodule.

The player manager coordinates scenario execution and data flow in the pipeline. It has scheduling and control duties.

Once a scenario is played, data is sent from one box to another in a unidirectional manner. Each box performs its processing job and then the scenario playback module is responsible for transferring the processed data to the next box.

3.4 Log and Error Management

Tracing and recording the workflow of events can be useful for different purposes (debugging, providing alerts, support). The kernel provides a convenient API for logging in the console or into a file through the log manager.

A specific manager is used to handle errors with the framework: the error manager.

3.5 Late-binding Configuration

Creating and playing a scenario are performed in a given context. A context is defined by the state of all configurable items (Kernel, plugins).

This context can be setup according to specific needs. Typically, a user may want to adapt the kernel log and error level according to the production phase (development, testing, and deployment).

The kernel provides a convenient API for late-binding configuration through the configuration manager.

The configuration manager reads configuration tokens from a file and initialize configurable items accordingly.

3.6 Summary

- ➢ End-user configure an execution context in a configuration file
- ➢ End-users create scenarios programmatically
- ➢ End-users run scenarios programmatically
- ➢ OpenViBE software system logs events at runtime either in console or in file

4. Technical Overview

As described in $2 System Purpose, OpenViBE provides its services through a set of C++ APIs based on core components.

4.1 Coding Language

The software system makes use of C++ that supports multiple paradigms. It relies on object-oriented inheritance and abstract virtual interface to achieve polymorphism. Templates are used where genericity is needed. Sometimes both are mixed as for implementing the mixin/parametrized inheritance pattern (see following view) widely used in the framework.

IV-CODESOURCE-MIXININHERITANCE

Primary Presentation

```
template <class T>

class Base : public T { ... };

class Derived : public Base<IDerived> { ... };

class IDerived : public IBase { ... };
```

Element Catalog / Description

The inheritance hierarchy is linearized: IDerived -> Base -> IDerived -> IBase

Rationale

The main goal is to simulate multiple inheritance and remove the risks bound to true multiple inheritance (see Diamond of dread).

4.2 Build Process

Build process management is handled with CMake v3.2.

4.3 Supported Platform

Platform	Version(s)	Compiler(s)
Windows	7/8/10	Visual 2013
Linux Ubuntu	14.04	Gcc 4.8

5. Software System Organization

In $3 Functional Overview, the Kernel module was introduced. A module is a logical subsystem part of the overall software system. The fact of splitting up a system into subsystem is called a breakdown.

The breakdown can be:

- Functional (identifying elements according to the set of coherent services they provide);
- Physical (identifying elements according to their physical representation/location);
- Both.

The Kernel module is a subsystem in a functional sense as it provides a set of management services but is also a subsystem in a physical sense as it resides in a specific location in the source code and is distributed as a single library.

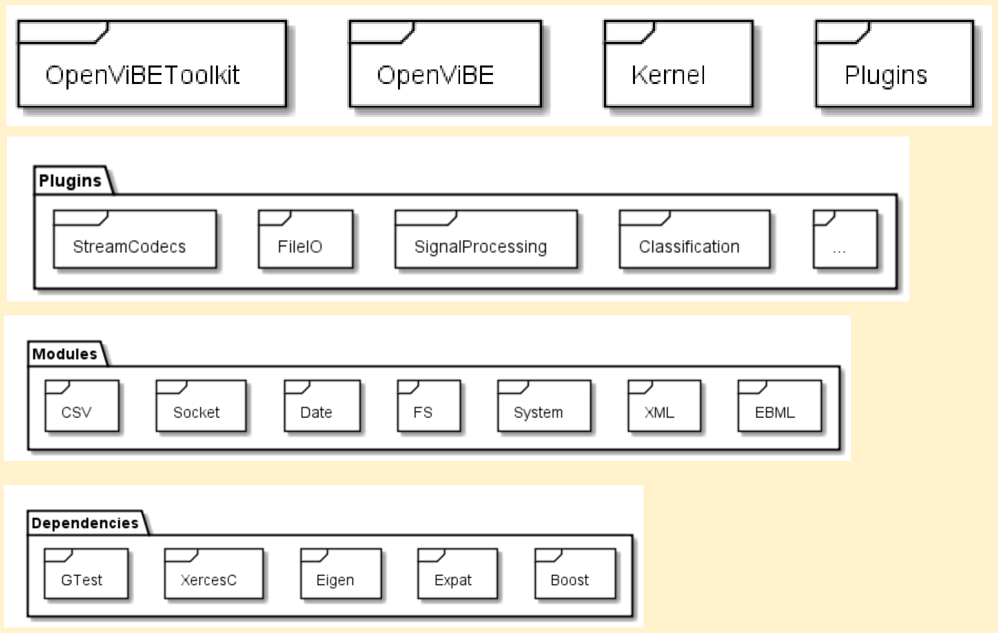
This chapter describes the system in a functional and in physical way. At the end, both perspectives are mapped together.

5.1 Functional Breakdown

The following view shows the functional breakdown of the system into modules.

LV-PackageDiagram-FunctionalBreakdown

Primary Presentation



Element Catalog / Description

This description gives a general overview of each module. Detailed description can be found in the MSD document. Dependencies between modules does not fill strong rules (no strict layering) except this one: a box in a plugin does not depend of a box in another plugin.

OpenViBE: Base framework that contains base classes.

Plugins: Extension mechanism that provides interfaces to create new boxes and algorithms**.**

Kernel: As described in $3 Functional Overview, the Kernel module provides a set of centralized management services.

OpenViBEToolkit: Helper module. Some services provided by other modules within OpenViBE are not easy to access or use. This module takes care of wrapping some of these complex services into a much more easy-to-use interface. It is a service usability facilitator.

Modules: Portable utility modules that provide some low-level services:

- Socket: Client-server network connection services

- Date: Date formatting and parsing services

- XML: XML data parsing and serializing services (based on Expat)

- System: System utility services related to memory consumption, timing etc.

- EBML: EBML data parsing and serializing services (see EBML Specifications)

- CSV: utility library used to load/save CSV data

OpenViBE Plugins: As described $3 Functional Overview, each plugin is a different module as it provides a set of coherent services. The diagram shows a sample of the existing plugins:

- Classification: Provides classification features (boxes and algorithms)

- Signal Processing: Provides signal processing features (boxes and algorithms)

- FileIO: Provides features to import/export files to include in a scenario (boxes and algorithms)

Dependencies: 3rd party libraries:

boost**:** C++ utility libraries (see Boost website)

- Version: 1.54

- Portability: Support gcc > 4.5 and MSVC > 8.0 SP1 on Windows XP/Vista/7

- License: Boost software license (usable in proprietary software)

- Library selection rationale: open-source, standalone, library developers in the C++ standard committee, some libraries transferred to the standard, formal review process (see boost review process).

expat: XML parser C library.

- Version: 2.1.0

- Portability: Multi-platform

- License: MIT license

- Library selection rationale: stable, lightweight, used in many open-source projects (Apache HTTP Server, Mozilla, PERL, Python) and originally developed by J. Clark (technical lead of the working group that developed XML – W3C).

Xerces C: XML parser C++ library with schema validation features (see xerces website).

- Version: 3.1.1

- Portability: supports gcc 4.8 and MSVC 2012

- License: Apache License 2.0

- Library selection rationale: standalone, standard (Apache), widely used, well documented.

eigen: High performance C++ linear algebra library (see eigen website).

- Version: > 3.1.1

- Portability: supports gcc > 4.1 and MSVC > 2008

- License: MPL2 (weak copyleft)

- Library selection rationale: open-source, standalone, cmake-based, reliable (see eigen reliability) widely used, well documented.

gtest: Unit test framework (see google test website).

- Version: 1.6.0

- Portability: Any c++98-standard-compliant compilers and MSVC > 7.1

- License: BSD 3-clauses

- Library selection rationale: open-source, standalone, cmake-based, widely used (OpenCV, LLVM, Chromium), well documented.

5.2 Physical Breakdown

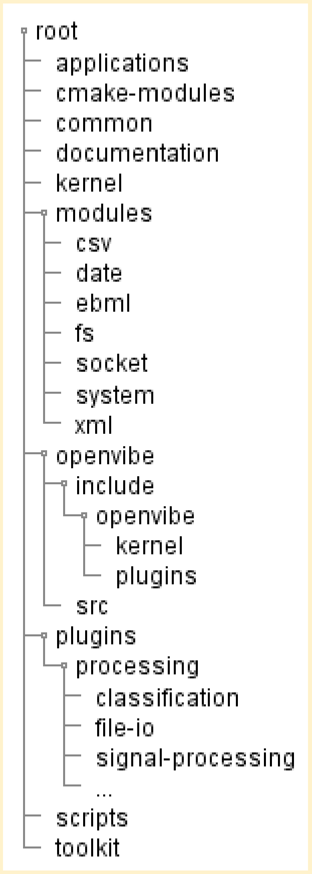
The physical breakdown can be performed by analyzing:

- The source code: a physical element is a file or a directory in that case;

- The binaries: a physical element is called a component and can be a library or an executable in that case.

PV-FileHierarchy-SourceCode

Primary Presentation



```
root
├ applications
├ cmake-modules
├ common
├ documentation
├ kernel
├ modules
│ ├ csv
│ ├ date
│ ├ ebml
│ ├ fs
│ ├ socket
│ ├ system
│ └ xml
├ openvibe
│ ├ include
│ │ └ openvibe
│ │   ├ kernel
│ │   └ plugins
│ └ src
├ plugins
│ └ processing
│   ├ classification
│   ├ file-io
│   ├ signal-processing
│   └ ...
├ scripts
└ toolkit
```

Element Catalog / Description

OpenViBE code source tree consists of the following directories:

applications: contains utility applications for developers and testers

- `openvibe-scenario-player` is used to play simple scenario or to execute test use cases described in a command file;

- `openvibe-id-generator` is used to generate unique identifier;

cmake-modules: contains cmake FindOpenViBEXXX modules used to import OpenViBE modules internally when a module depends on another one, cmake FindXXX modules for external dependencies, other cmake helper scripts.

common: contains some common include files defining common type that should be used within the framework (`ov_common_types.h`), common utility preprocessor definition (`ov_common_defines.h`) and shared box identifiers (`ovp_global_defines.h`, see MSD for details on identifiers).

documentation: contains all script and configuration files to generate doxygen documentation.

kernel: contains Kernel default and only implementation. It contains the implementations of all abstract interfaces defined in `include/kernel` subdirectory of `openvibe` directory with some additional classes necessary to the implementation.

modules: contains one directory for each low-level modules as defined in LV-PACKAGEDIAGRAM-FUNCTIONALBREAKDOWN.

openvibe: conatains APIs abstract interfaced and base framework:

- `include/openvibe/kernel` directory contains all kernel-related abstract interfaces that any kernel implementation must implement

- `include/openvibe/plugins` directory contains box and algorithm abstract interfaces that must be implemented within plugins to implement a box or an algorithm.

- `include/openvibe` and `src` directories contain framework base classes and interfaces.

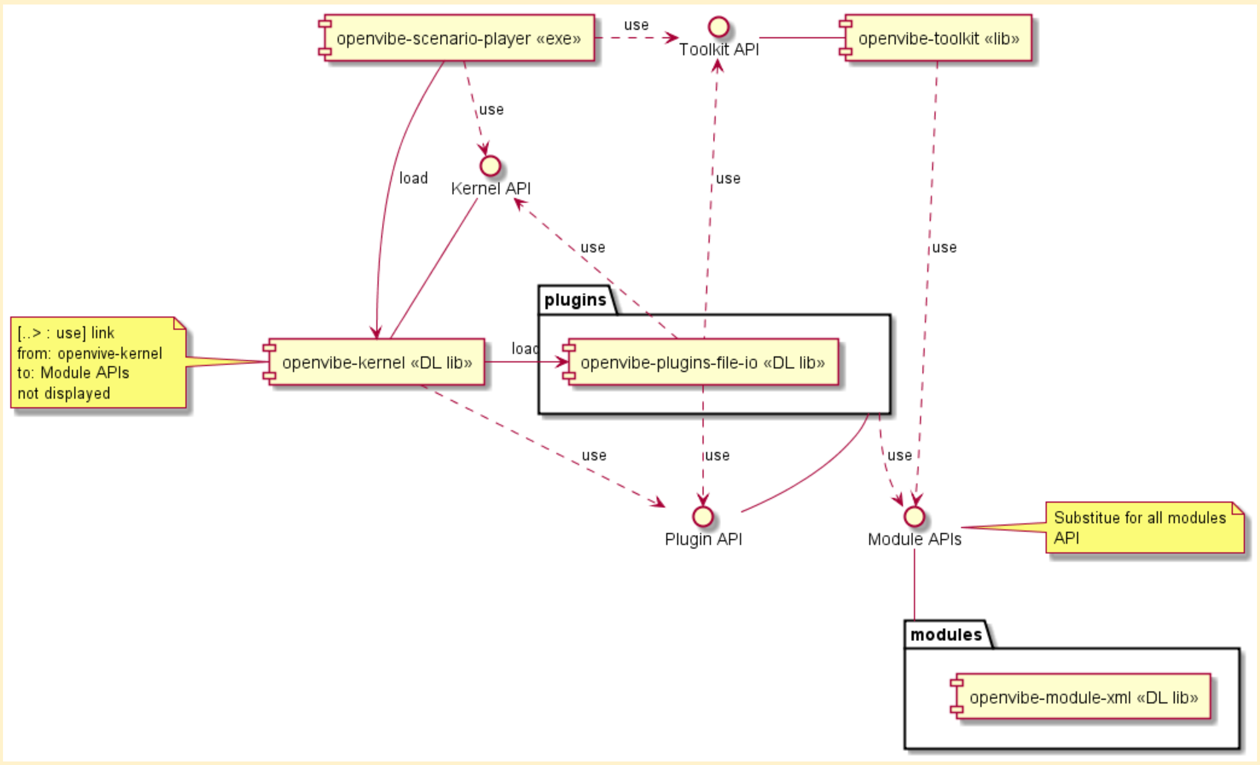
plugins: contains all plugins. For instance `plugins/processing/classification` contains implementations of all boxes and algorithms related to classification.

scripts: contains build and install scripts.

toolkit: contains implementation of all toolkit services.

PV-ComponentDiagram-Binaries

Primary Presentation



Element Catalog / Description

<<lib>> : static library (.a/.lib) OR shared/dynamic library (.so/.dll) dynamically linked at runtime but statically aware (the libraries must be available at compile/link time.

<<DL lib>> : shared/dynamic library loaded/unloaded at runtime programmatically using the loader system function.

<<exe>>: Executable.

—— (No caption plain line): Means the component provides the API.

In this diagram, the scenario player application is chosen as end-user application.

To differentiate this perspective from the functional one, the term component is used to refer to an element instead of module.

The application has to load the `openvibe-kernel` component at runtime. It then uses the `Kernel API` to take advantages of Kernel services (creating/running pipeline). The `openvibe-kernel` component is responsible for loading the plugin components at runtime (a single plugin is shown for the sake of clarity). Each plugin can contain boxes and/or algorithms that are used through the `Plugin API` (different interfaces are used for boxes and algorithms).

A plugin can use the `openvibe-toolkit` component and the `openvibe-kernel` component through their API. Moreover, it can use the `Plugin API` in some specific cases: a box manipulating an algorithm or an algorithm manipulating another algorithm (**but a box manipulating a box is forbidden**).

All components can use low-level modules. The diagram just shows one of the existing module component but each component provides its own API.

Notes

The base framework component `openvibe <<lib>>` is not shown for the sake of clarity but can be used by the kernel, the toolkit and the plugins components.

Dependencies components are not shown on the diagram.

- `expat <<lib>>` is used by module openvibe-xml-module.

- `xerces-c <<lib>>` is used by FileIO plugin.

- `boost <<lib>>` can be used by any component.

- eigen and some boost components are header-only.

5.3 Mapping

The following table presents the mapping between functional and physical elements. FileIO plugin has been chosen as an example for the mapping of plugins but the behavior is the same for all plugins.

Functional Module	Binary Component	Source Code File/Directory (relative to OpenViBE root unless otherwise stated)

OpenViBE Base framework	openvibe	`./openvibe/include/openvibe`
		`./openvibe/src`

Plugins	*NA: header-only*	`./openvibe/include/openvibe/plugins`

Kernel	openvibe-kernel
	dynamically loaded
		Interfaces:
		`./openvibe/include/openvibe/kernel`
		Implementation:
		`./kernel`

OpenViBEToolkit	openvibe-toolkit	`./toolkit`
Date	openvibe-module-date	`./modules/date`
EBML	openvibe-module-ebml	`./modules/ebml`
FS	openvibe-module-fs	`./modules/fs`
Socket	openvibe-module-socket	`./modules/socket`
System	openvibe-module-system	`./modules/system`
XML	openvibe-module-xml	`./modules/xml`
CSV	openvibe-module-csv	`./modules/csv`
FileIO *(plugin example)*	openvibe-plugins-file-io *dynamically loaded*	Interfaces for box and algorithms:

		`./openvibe/include/openvibe/plugins`
		Implementation:
		`./plugins/processing/file-io`

Eigen	*NA: header-only library*	Windows:
		`./dependencies/eigen`
		Linux (relative to system root):
		`/usr/include/eigen3`

Expat	libexpat	Windows:
		`./dependencies/expat`
		Linux (relative to system root):
		`/usr/include/expat*.h`

Boost	libboost_xxx
	(compiled components)
		Windows:
		`./dependencies/boost`
		Linux (relative to system root):
		`/usr/include/boost`

XercesC	libxerces-c	Windows:
		`./dependencies/xerces-c`
		Linux (relative to system root):
		`/usr/include/xercesc`

gtest	Libgtest	Windows:
		`./dependencies/gest`
		Linux (relative to system root):
		`/usr/include/gtest`

6. Appendices

6.1 Architectual View Template

IDENTIFIER *// ViewType_DiagramType_ID*

Primary Presentation // Main Diagram

Element Catalog / Description // Diagram legends and description

Package1: This package is responsible for…

Package2: This package is responsible for…

…

Notes // Note on the view (why this element was eluded etc.)

Rationale // Architectural decision justification

SRS Requirement number XXX *// Reference to requirements*

This decision was made for maintainability. // *Explanatory sentence*