DPF Manager

The open source COMMUNITY

# DPF MANAGER

**TECHNICAL SPECIFICATIONS DOCUMENT**

Project acronym: PREFORMA

PREFORMA - Future Memory Standards

PREservation FORMAts for culture information/e-archives

EC Grant agreement no: 619568

EC Call ID: FP7-ICT-2013-11

# INDEX

# 1. Document Overview

## 1.1. Purpose

This document describes the technical decisions and final architecture of the DPF Manager in order to fulfil the requirements and features described in the Functional Specifications Document.

It provides an overview and a detailed explanation of the proposed architecture, data flows, data structures and external interfaces (APIs and Graphical User Interface).

## 1.2. Document structure

Section 1 provides the purpose, the structure of this document as well as references used in the preparation of this document and a list of abbreviations.

Section 2 provides an introduction to the DPF manager tool.

Section 3 introduces the architecture and the main components of the DPF manager as well as their interaction.

Section 4 describes the in detail the architectural components, modules and how decoupled they are.

Section 5 describes all how the different components interact with each other by means of data exchange.

Section 6 describes the design and structure of the data shared between the different modules in the architecture.

Section 7 justifies the choice of program language, minimum required version and third party libraries.

Section 8 explains the benefits of the proposed architecture.

Appendix A includes a high-level architecture diagram.

Appendix B includes a table with all the 3rd party libraries we plan on using, together with their licenses and linking strategies.

## 1.3. Project References

This Section provides a list of the references that were used in preparation of this document:

TIFF Revision 6.0 Final — June 3, 1992

Adobe Photoshop® TIFF Technical Notes March 22, 2002

Adobe PageMaker® 6.0 TIFF Technical Notes September 14, 1995

ISO 12234-2:2001 Electronic still-picture imaging -- Removable memory -- Part 2: TIFF/EP image data format

ISO 12639:2004 Graphic technology -- Prepress digital data exchange -- Tag image file format for image technology (TIFF/IT)

*ICC.1:2010 Image technology colour managements – Architecture, profile format, and data structure.*

XMP specification part 1 - Data model, serializations and core properties. April, 2012

XMP specification part 2 - Additional properties. April, 2012

XMP specification part 3 - Storage in files. May 2013

Exchangeable image file format for digital still cameras: EXIF Version 2.3. December, 2012

Dublin Core Metadata Initiative, August 2007

IPTC Standards - Photo Metadata White Paper, 2007 Document Revision 11

XML Schemas (XSD) Reference

Design Principles and Design Patterns, Robert C. Martin, 2000

Principles Of OOD , Robert C. Martin

OSGi Core Release 6, June 2014

*PREMIS Data Dictionary for Preservation Metadata, July 2012*

*METS METADATA ENCODING AND TRANSMISSION STANDARD: PRIMER AND REFERENCE MANUAL , 2010 Digital Library Federation*

## 1.4. Acronyms and Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **DPF** | Digital preservation files |
| **EXIF** | Exchangeable image file format |

| | |
|---|---|
| **GLAM** | Galleries, Libraries, Archives and Museums |
| **ISO** | International Organization for Standardization |
| **OAIS** | Open Archival Information System |
| **PREFORMA** | PREservation FORMAts |
| **PDF** | Portable Document Format |
| **OSGi** | Open Service Gateway initiative |
| **URL** | Uniform Resource Locator |
| **XML** | eXtensible Markup Language |
| **XMP** | eXtensible Metadata Platform |

# 2. Introduction

DPF Manager is an application and a framework designed to allow end users and developers to gain full control over the technical properties and structure of digital content Data Objects intended for Long Term Preservation.

In considering the suitability of particular Data object for the purposes of preserving digital information as an authentic resource for future generations, relies on the use of a stable, open and well documented file format as well as some data object properties acceptance criteria.

The main objective is to give memory institutions full control of the process of the conformity checks of files. This is a four-step process:

- Identification: the process of determining the file format of a Data Object based on the file extension and the file signature.
- Validation: the process of determining the conformance to a specific normative. These normative can be defined by some standardization organization or specific acceptance criteria based on a locally-defined policy rules.
- Modification: the process of modifying the Data Object, preserving the Information Representation, in order to make it more suitable for long term preservation.
- Reporting: the process of collecting and submitting the data object structure and metadata as well as validation result with the modification information.

DPF Manager provides the tools to process a large number of files from different sources completely automatically. The internal architecture is flexible enough to be suitable for multiple scenarios.

The DPF manager architecture was not only designed to fulfil the functional and operational requirements, it was also designed to make it really easy to extend and integrate into other systems, to establish a sustainable developer community around the framework.

#  3. Architecture overview

This section provides an overview of the architecture principles and the main components of the DPF manager. The detailed architecture components, data structures, data flows and APIs are described in detail later in other sections of the document.

## 3.1. Introduction

Establishing a stable and active community of developers around the open source DPF Manager is a mandatory requirement in this project; consequently the architecture has been designed with the following principles in mind:

- Simplicity: The proposed architecture follows KISS design principle acronym for "*Keep it simple, stupid*". Developers really appreciate simple designs with fast learning curves that allow them to focus their efforts in developing rather than learning complicate architectures.

- Use of standards: All the Architecture follows standard design patterns. Design patterns provide the architecture with well-tested solutions to common programming problems improving many aspects of the system. In addition, using widely known design patterns encourages more legible and maintainable code helping the developers understand the architecture.

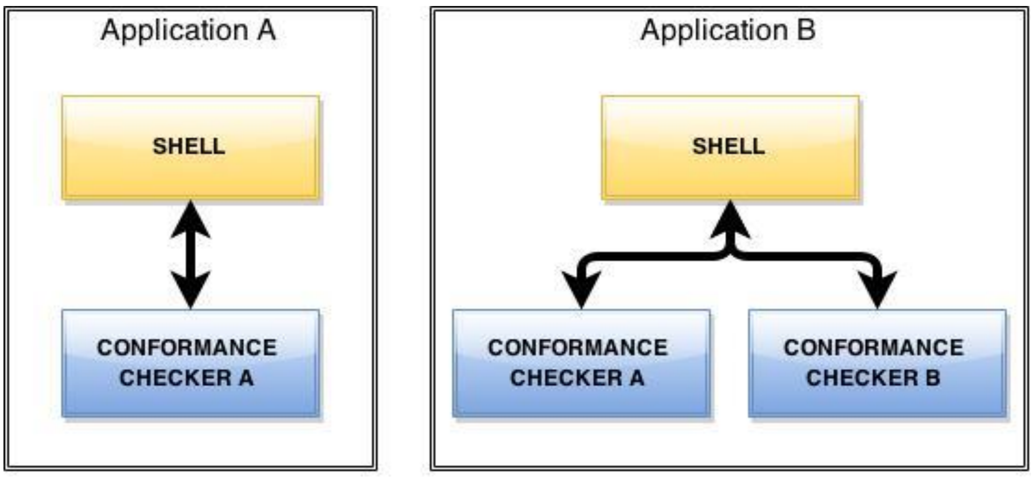
- Modularity: The application has been divided into a set of loosely coupled functional units, named modules, that represent a set of related concerns. Modules are independent of one another but can communicate with each other in a loosely coupled fashion. Using a modular architecture design makes it easier to develop, test, deploy, and maintain the application.

- Decoupled: To ensure the module independence, an event driven architecture is proposed. Event driven architectures have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows. The software development proactor pattern is going to provide this functionality.

- Conformance Checker independence: The architecture has been designed to decouple the Shell from the Conformance Checker. The Shell has no initial knowledge about the capabilities implemented by the Conformance Checker. When the Shell receives a request via one of its multiple interfaces, it queries the Conformance Checker via an internal API. The Conformance Checker responds with the list of standards it can validate against, the set of rules it can accept as part of the internal acceptance criteria, the fixes that it can apply to increase conformance, and the reports it can produce. Developers will not have to make any modification in the shell when they implement new Conformance Checkers or create new functionalities.

## 3.2. Components

The DPF manager architecture has been separated in two main components, the Shell and the Conformance Checker.

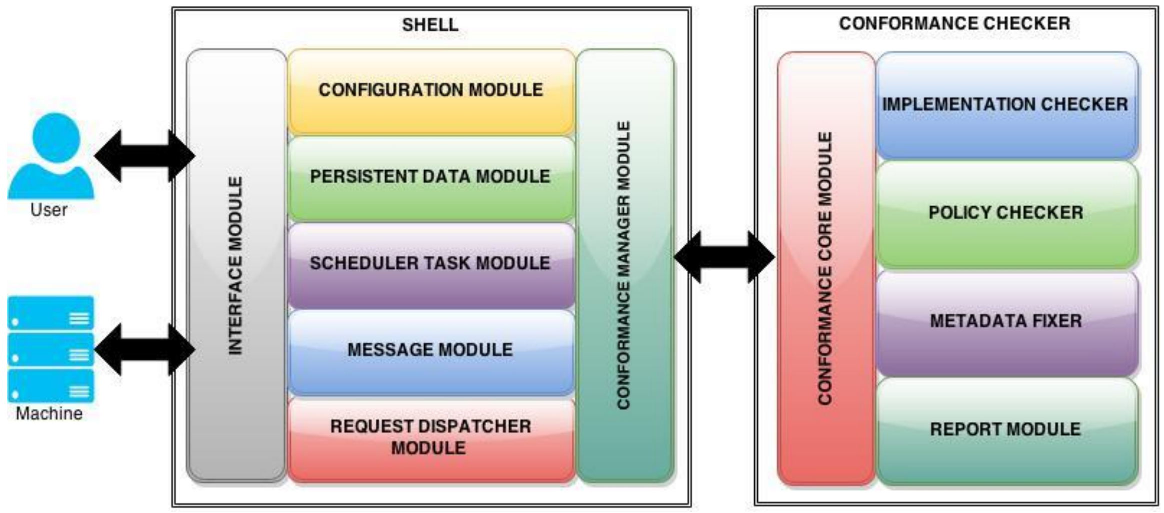


**Picture 1 DFP Manager Components.**

DPF manager is built with an implementation of a Conformance Checker for TIFF files but the architecture is designed to be able to interact with other implementations at the same time as show in the picture.

Each Component in the architecture has been organized in different modules. Users and other systems interact with the Conformance Checker via the Shell component. The rest of the modules are "hidden" behind it, in a process that is transparent to the users.

The following diagram gives an overview of the architecture of the system:



**Picture 2 DPF manager modules.**

### 3.2.1. Shell

The Shell provides the interface between the users (humans, other systems via APIs, etc.) and the Conformance Checker/s.

The Shell implements three different external interfaces to support all the possible scenarios detailed in the PREFORMA requirements:

- A command line client
- A network service that can accept requests via sockets and/or a HTTP/S REST API.
- A Graphical User Interface (GUI), that will guide expert and non-expert users alike through the whole conformance checking process, from the configuration settings to the displaying of the report

In the process of considering the acceptance of particular Data Object or a collection of data objects for the purposes of Long Term Preservation the Shell requires the location where the data is stored and a configuration XML file with all the parameters that are going dictate the behaviour of the system.

Then the Data Object format is identified and the file is sent to the appropriate Conformance Checker, if it exists (it could happen that the user tries to validate a format that is not accepted by any Conformance Checker).

During the checking process the Shell needs to provide information about the actions that the system is going through as well as logging all the activity.

All the information generated during the process need to be stored for futures access.

As the Conformance Checker works with one file at a time, the Shell is able to generate a global report that aggregates the results of each single file.

### 3.2.2. Conformance Checker

When a Data Object is checked, first of all the file is read and the internal structure is parsed in an Object Representation, that contains its structure (TIFF files do not have a flat structure, the information can be stored at multiple levels) and all the metadata information.

Then, this structure is validated against a standard and the rules defined by the user in the policy checker. The standard and the rules are defined in the XML configuration file.

Once the result of the validation is known, some minor fixes can be automatically applied to the Data Object. The only pre-condition for these fixes is that they should always preserve the Information Representation in order to improve the sustainability of the file (that is, the fixes can be applied to the metadata but never to the image data itself).

Finally, a report (in XML of JSON format) with the file Representation Information, file metadata, and validation and fixer results is send to the Shell.

### 3.2.3. Shell - Conformance Checker interactions

**Initialization**

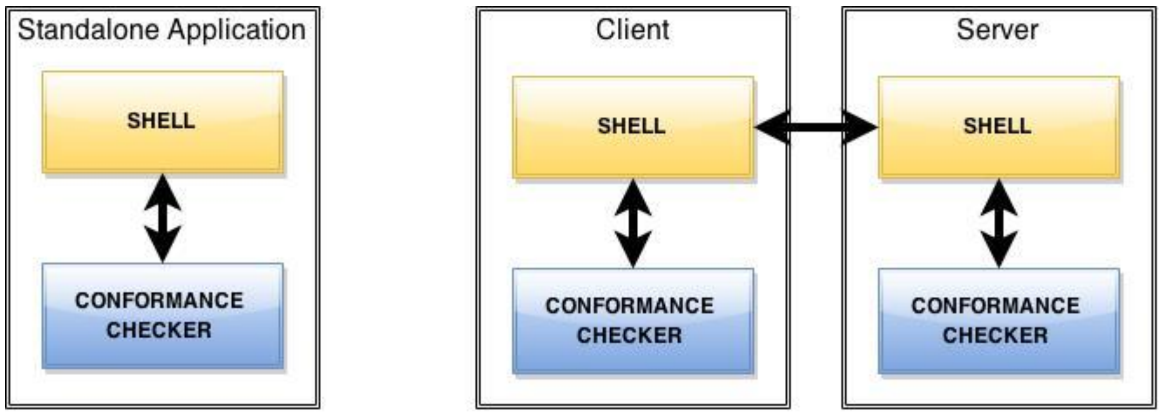
One of the principles of the architecture is the aim to decouple the Shell form the Conformance Checker. Therefore, the Shell has no previous knowledge about:

- What formats the Conformance Checker is able to read.
- How to identify the files appropriate for a Conformance Checker.
- What standards can be validated by the Conformance Checker.
- What rules can be used to define the local acceptance criteria.
- What fixes can be applied to a Data Object.
- How to validate the input XML configuration file.

So when the application is initialized the Shell asks the Conformance Checkers for all this information.

This discovery mechanism is extremely useful when working with multiple Conformance Checkers from different suppliers, as the Shell can interact with them and find out what they are able to do without requiring any specific configuration (that has to be managed, can change over time, and therefore represents an increase in the maintenance costs).

This process can differ if the application is configured in standalone mode (where the Shell queries the built-in Conformance Checker) or in a the Client- Server implementation where the Shell acting as a client asks the Shell acting as a server about his Conformance Checker capabilities.



Picture 3 DPF Manager Standalone and Client-Server implementations.

The following picture illustrates both cases. As show in the Client-Server implementation the communication information flow between the Client-Server and the Shell-Conformance is the same.

This is an important point in the architecture design, as we have decided to use the same communication protocol with the same information flow in both cases. This simplifies the architecture and at the same time it provides the flexibility and horizontal scalability needed in the platform.

This decoupled modularity will make it easier for third party developers to understand how the system works and to build on top of it.

**Validation process**

After obtaining the information about the Conformance Checker capabilities, the Shell knows how to identify which files are appropriate for which Conformance Checkers,  so it is ready to start accepting and validating files.

During the validating process the Conformance Checker continually informs the Shell about the steps it is undertaking (opening the file, extracting metadata, validating against TIFF/IT, etc.).

This information can be logged and/or shown to the end user (checking a large number of files could take a noticeable amount of time, so the system keeps the user informed about the progress made).

Finally, when a file is validated the results are returned to the Shell.

# 4. Architecture components

## 4.1. Introduction

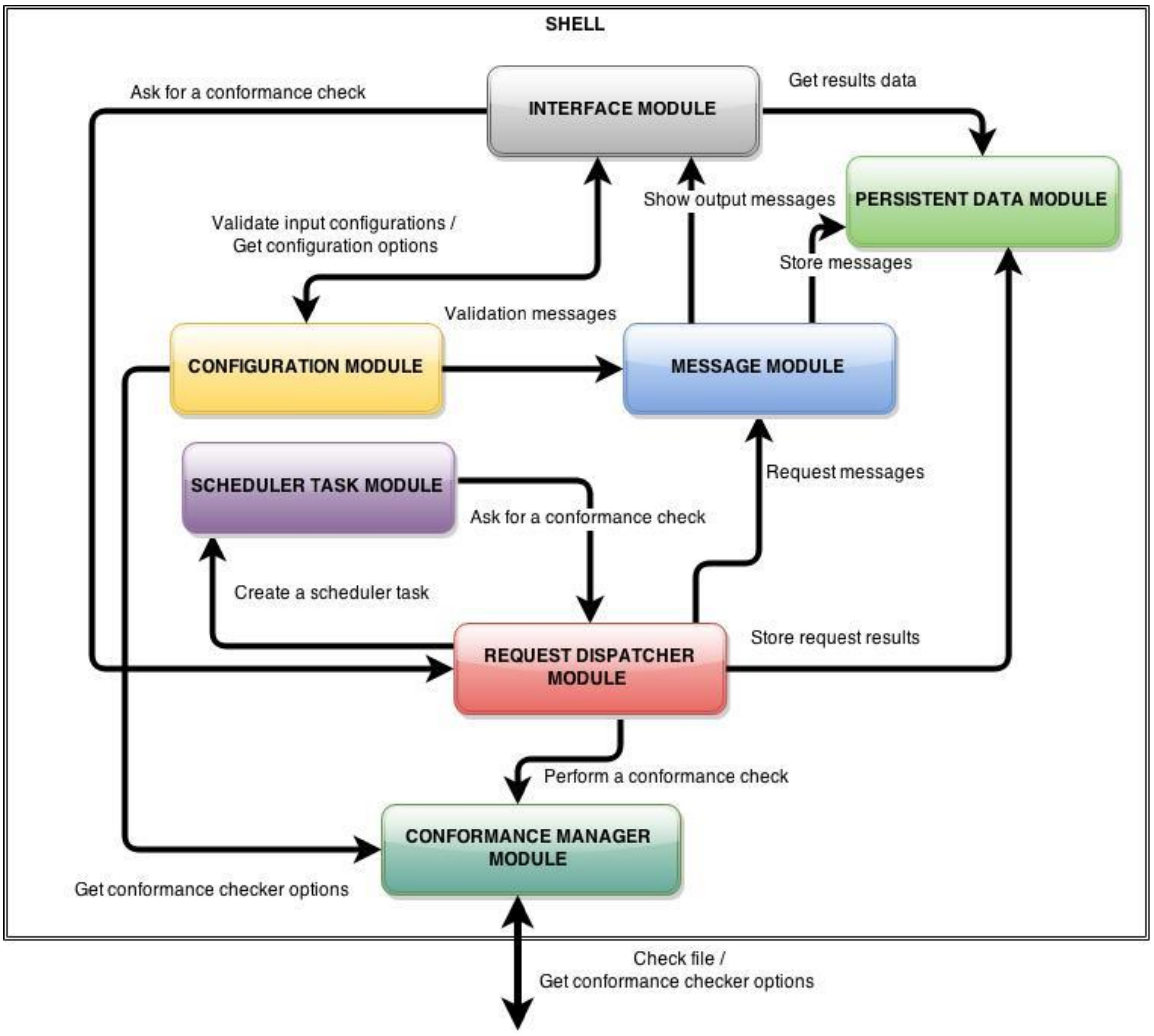
A brief overview of the system components was given in section 3 Architecture overview. This section provides further information and the internal architecture for each component.

## 4.2. Shell

The internal Shell architecture has been designed in a modular way. Grouping all the shell functionalities in different modules will help developers understand the architecture as well as contribute new code. The Shell modules are:

- Interface module: management of user or system interactions.
- Persistent data: responsible for storing persistent information.
- Message module: control and distribution of all the application messages.
- Configuration module: manages the configuration and validation of configuration files.
- Scheduler task module: control of the scheduled checks.
- Request dispatcher module:  manages the input files and output reports.
- Conformance manager module: manages the communication with the built-in and remote Conformance Checker/s.

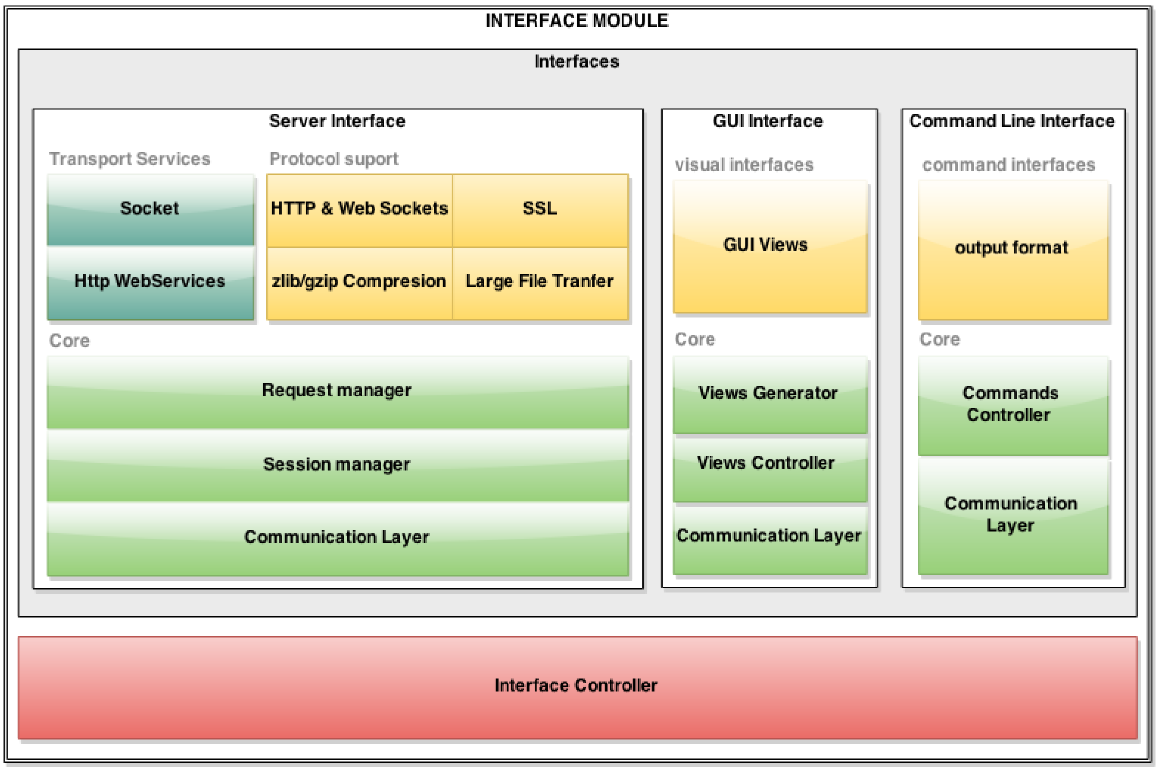
In the following sections all these modules functions and architecture is explained in detail as well as the interaction with the other modules.


SHELL
INTERFACE MODULE
Ask for a conformance check
Get results data
PERSISTENT DATA MODULE
Show output messages
Validate input configurations / Get configuration options
Store messages
Validation messages
CONFIGURATION MODULE
MESSAGE MODULE
SCHEDULER TASK MODULE
Request messages
Ask for a conformance check
Create a scheduler task
Store request results
REQUEST DISPATCHER MODULE
Perform a conformance check
CONFORMANCE MANAGER MODULE
Get conformance checker options
Check file / Get conformance checker options

Picture 4 Shell modules and interactions.

### 4.2.1. Interface Module

The Interface Module has the responsibility to inform about the Conformance Checker capabilities, receive the input files and configuration to perform a conformance check, inform about the current process as well as provide access to the results.



Picture 5 Interface Module architecture.

The interface module includes three interface implementations to fulfil all PREFORMA required scenarios.

### Server interface

This interface implements the server communication. When the shell is configured to work as a server this implementation is invoked to receive requests from clients.

This implementation provides all the tools needed in a Client-Server communication through a network, supporting different protocols, providing SSL security, Large File Transfer capabilities as well as Zlib/Gzib Compression.

The server can dispatch the requests from multiple clients at the same time using the session manager layer.

### GUI interface

This implementation provides a graphical user interface to interact with the Conformance Checker. This interface follows the standard Model - View - Controller pattern architecture. This pattern is used to separate application's concerns:

- Model - The communication Layer update the Views Controller if its data changes.

- View - GUI View represents the visualization of the data that model contains.
- Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

The view generation component is responsible to generate the appropriate view depending on the Conformance Checker options and functionalities, providing a user friendly interface to create a configuration file.

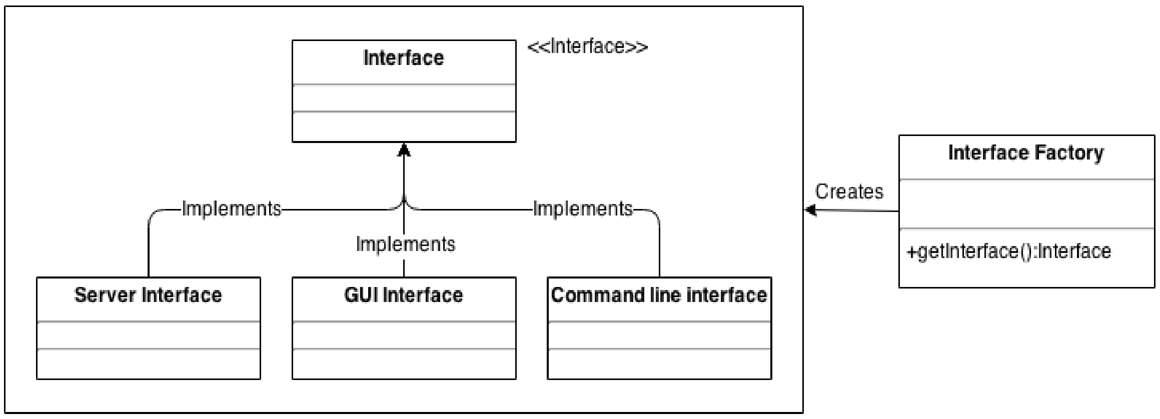
This component exposes all the functionalities developed inside a Conformance Checker to the user.

**Command line interface**

When the DPF manager is used as a command line interface this interfaces provide all the commands necessary to interact with the Conformance Checker as well as show all the process feedback.

**Interface object**

All this interfaces implements the abstract interface class. The DPF manager invokes the appropriate interface using the Interface Factory.



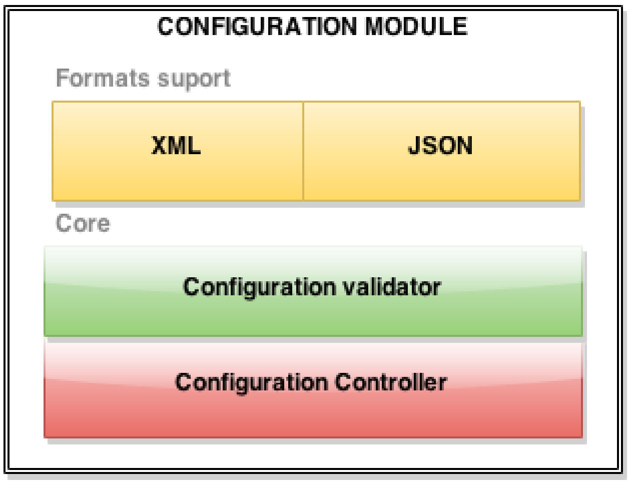
Picture 6 Interface object factory design pattern.

The design Factory Pattern will help the developers creating new interface, implementing the abstract class interface and modify the Interface Factory method getInterface().

The Interface controller component is the responsible of the communication with the other modules.

### 4.2.2. Configuration Module

The configuration manager have the responsibility to get the Conformance Checker options, validate the input configuration files and report if there is an error with the validation. The configuration can be provided using a XML or JSON format.



Picture 7 Configuration Module architecture.

**XML configuration**

The XML validation process is done using the standard XML schema commonly known as an XML Schema Definition (XSD).

XSD formally describes what a given XML document can contain, defines the shape, or structure, of an XML document, along with rules for data content and semantics such as what fields an element can contain, which sub elements it can contain and how many items can be present. Moreover, it can also describe the type and values that can be placed into each element or attribute.

The module contains a general XSD with the general structure of the configuration file. In the initialization process the module ask the Conformance Manager module about the Conformance Checker options. Then, update the general XSD with the Conformance Checker options in order to validate the XML configuration files.

**JSON configuration**

If a JSON configuration file is used then there is a JSON to XML conversion in order to validate the configuration. This process is mandatory because the lack of an established standard way to validate a JSON structure.

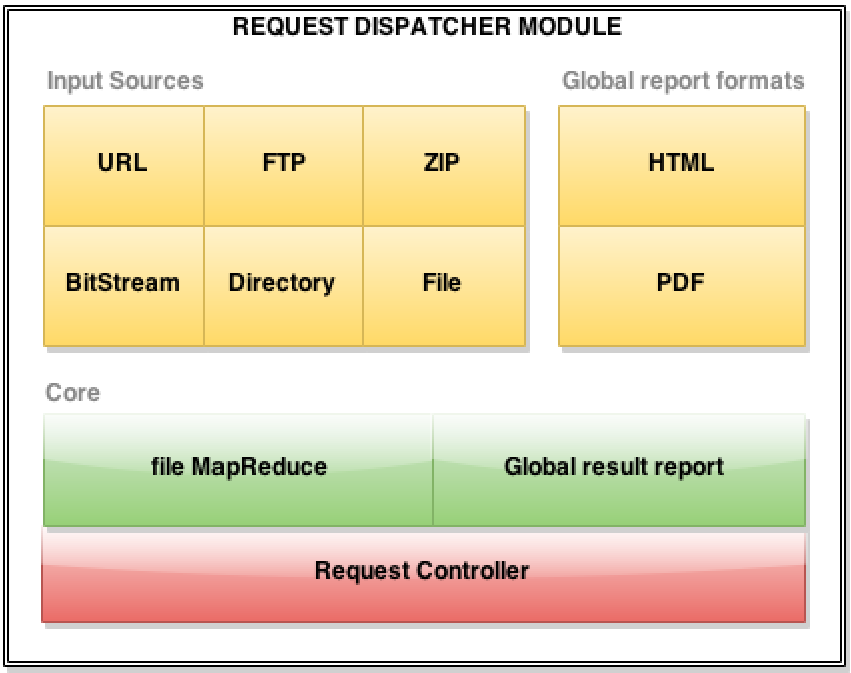
The validation result is send to the Message Module.

### 4.2.3. Request Dispatcher Module

The request dispatcher module is the responsible to attend the Conformance Checker requests. This module is able to interpret different source of inputs and insert the files in a MapReduce container. This container is send to the Conformance Manager to process the files.

The configuration could require executing a scheduled schedule at a specific, in this case this module is the responsible to inform and configure the scheduler task module.

This module is also responsible to process all the messages received from a check as well as storing the results.

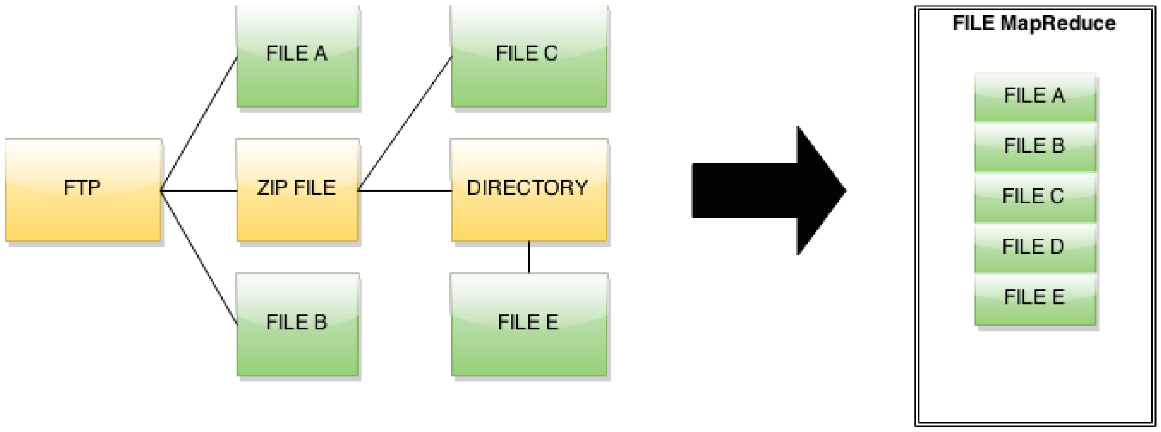


Picture 8 Request Dispatcher Module architecture.

**Input sources**

This module is the responsible to process the input sources and fill the file MapReduce object.

The input source might be a file, a bitstream, directory, URL or FTP location and is treated as the root node of a hierarchical tree. This root node may encapsulate other subsidiary source units, this source units are treated as children nodes. This process is repeated for each subsidiary input source found.

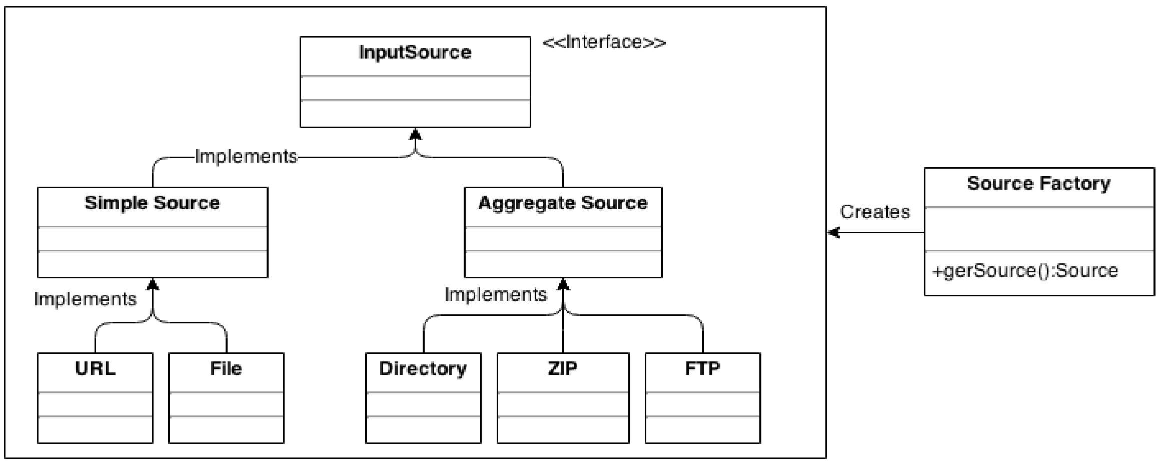


Picture 9 Source Input treatment.

When this hierarchical tree is completed is ready to be transform into the MapReduce object.

**Input source Object**

Developers can implement new input sources whenever they need, using the abstract class Simple Source or Aggregate Source depending if the source is a file container or a single file and changing the getSource() method in the Source Factory.



Picture 10 Input Source Object inheritance.

**Global report**

This module generates a global report with a general overview of all files validated and a specific report for each file. All the reports (in XML or JSON format) generated by the Conformance Chequer are collected for extract general statistics of the process.

- Global Conformance Checker result.
- Global Implementation Chequer result.
- Global Policy Checker result.

- Number of errors
- Number of warnings
- Errors fixed
- Most common errors
- Most common warnings

It also generates a single report from each file with:

- The internal file structure.
- Metadata inside the file.
- Link to the resource.
- Results of the Conformance Checker.
- Result of the Implementation Checker.
- Result of the Policy Checker.
- Result of the Metadata Fixer.

This general report may be generated in two formats: an easy to share and store report in PDF format and an interactive report in HTML format.

All these reports are generated using generic templates. These templates are filled with the information and then are composed in order to generate the final report.

**Similar templates**

- Summary template: has to be filled with the general statistics and composed with the result of the Summary single template for each file test result.
- Summary single template: has to be filled with the information generated by the Conformance Check.
- Single report template: has to be filled with the information generated by the Conformance Check.

**PDF specific templates**

- Cover template: Define the style and content of the report first page.
- Table of content template: Define the style or index page the content of this page is auto generated.
- Back page template:  Define the style and content of the report last page.
- PDF template: Define the main PDF layout.

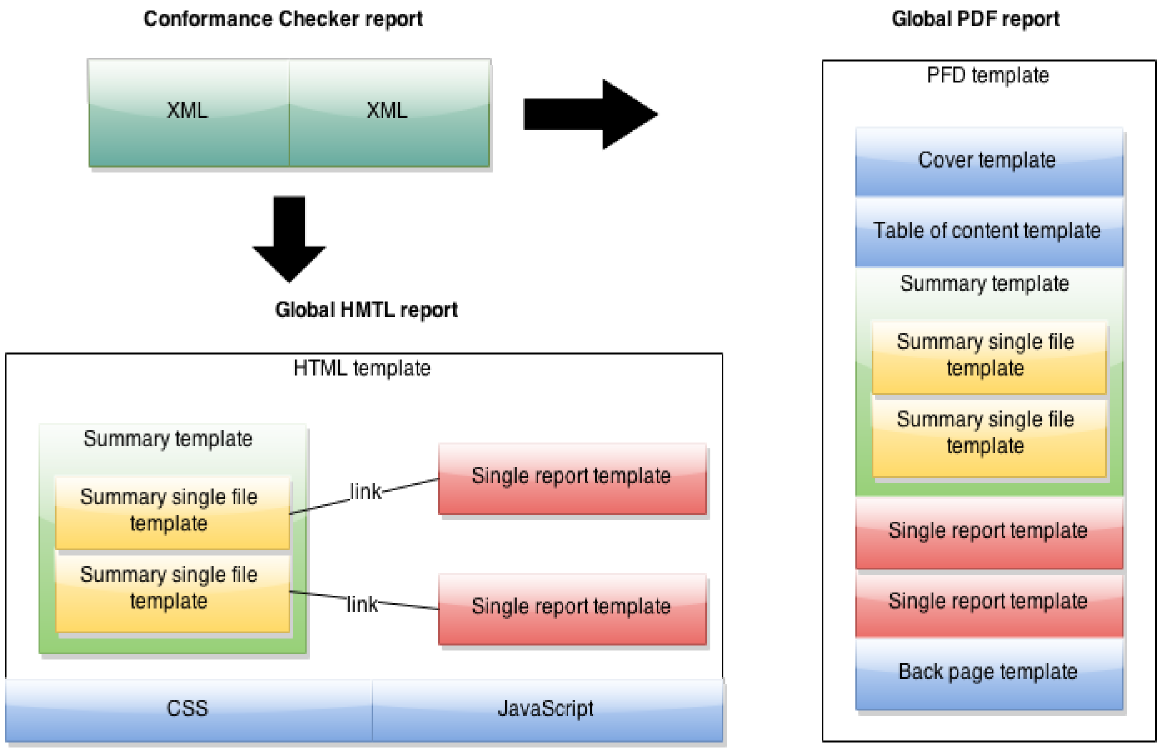
**HTML specific templates**

- HTML template: General main HTML page layout, this layout import the CSS and JavaScript script as well as the search and navigation options.

In the PDF report all these templates are in XML format following the PDF standard, composing all the templates inside the PDF template results in the final report file.

On the other hand, in the HTML report all the templates follows the HTML format and in this case  the summary template acts as the first page and the Single report templates are HTML pages linked in for the main page.

The following picture shows an example of the two reports generated as results of two files validation.

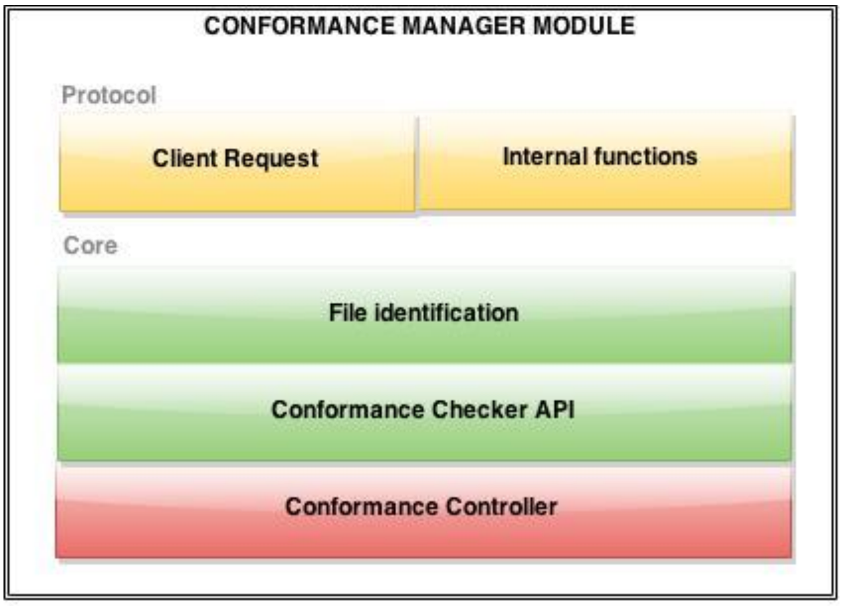


Picture 11 Global report construction.

Working with templates, is the easiest way to enable the report customization.

### 4.2.4. Conformance manger module

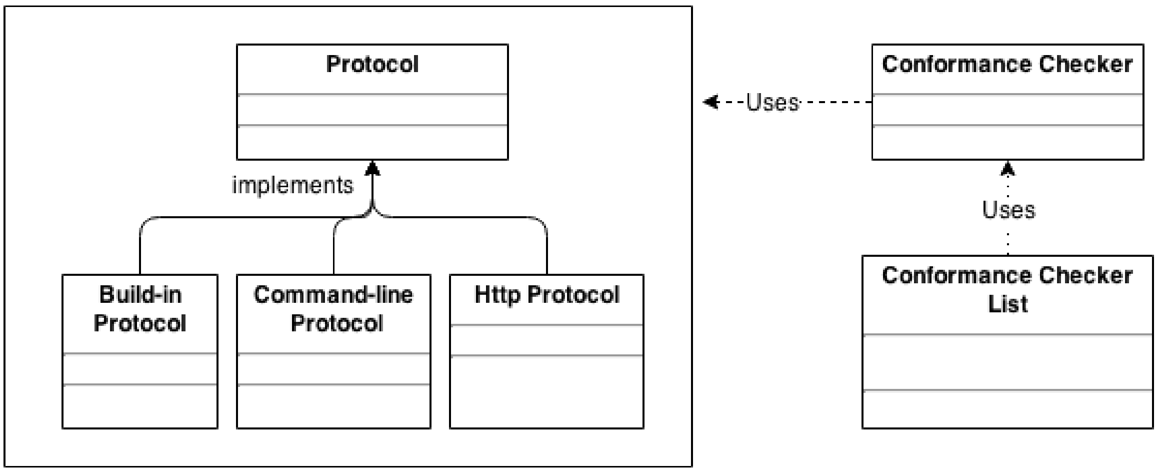
The conformance manager module is responsible of identify, get the configuration and the management of the conformance checker/s attach to the shell.



Picture 12 Conformance Manager Module architecture.

During the initialization this module has to detect all the Conformance Checkers implemented inside the DPF manager. Then it can initialize these Conformance Checkers and attach them to the Conformance Checker List.

If the configuration XML file requires a remote connection to another Shell (Client - Server connection or command-line connection) then this module asks for the remote available Conformance Checker/s and attach this new Conformance Checker/s to the list using the appropriate protocol.



Picture 13 Conformance Checker object structure.

At this time, the shell is ready to ask to each Conformance Checker in the list about all functionalities that the developers have implemented as well as the way to identify the files appropriate for this Conformance Checker

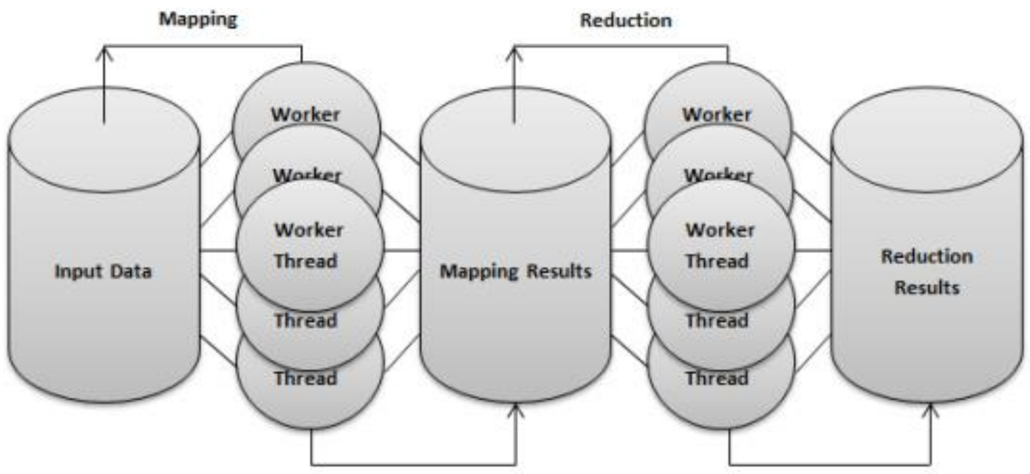
**File identification**

This module is able to identify a file and chose the correct Conformance Checker for it. This file identification is done using the file extension and the Magic number.

*The Magic numbers are a group of bits of a file with a concrete value which uniquely identify the type of file*.

This makes programing easier because complicated file structures need not be searched in order to identify the file type. The file identification is crucial in the Client- Server implementation, the client knows if the remote shell is able to validate the file without having to send the file through the network.

The file validation is done with the file MapReduce container, this design pattern let us identify a large number of files in a multithreaded environment and obtaining a new map with the files that has to be check for a concrete Conformance Checker.

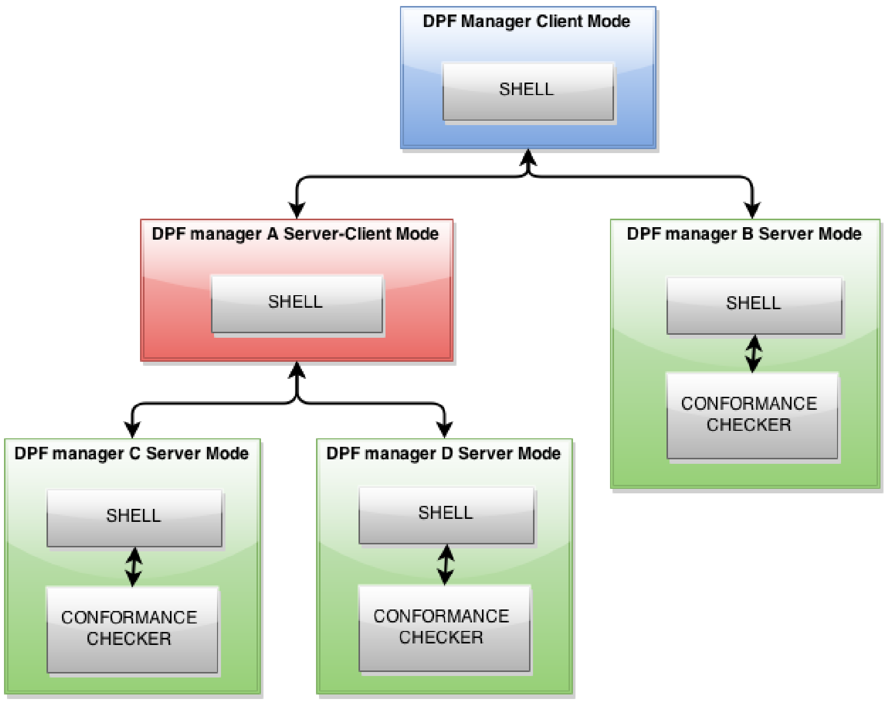


Picture 14 Map Reduce design pattern.

Now the DPF manager is ready to validate files, for each Conformance Checker object in the list We apply a reduce function with his validation function. All the files from the reduction results are send to this Conformance Checker independently if it is in a remote DPF manager or build-in.

**Remote test**

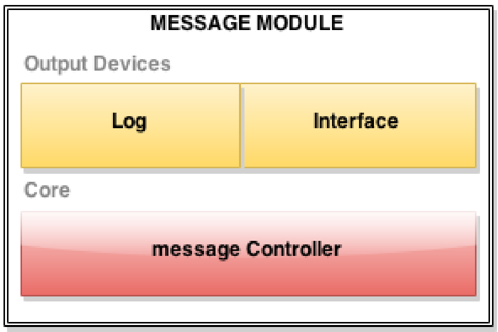
This module functionality enable the horizontal scalability of the DPF Manager and can create complex structures as show in the next image where the DPF manager A is working in a Server-Client configuration distributing the checks between two servers.


DPF Manager Client Mode
SHELL
DPF manager A Server-Client Mode
SHELL
DPF manager B Server Mode
SHELL
CONFORMANCE CHECKER
DPF manager C Server Mode
SHELL
CONFORMANCE CHECKER
DPF manager D Server Mode
SHELL
CONFORMANCE CHECKER

**Picture 15 Scalability DPF manager sample.**

### 4.2.5. Message module

This module is responsible for controlling all messages received from different components in the shell and redirecting the messages to the correct module depending if the message has to be sent to the interface or stored in a log.

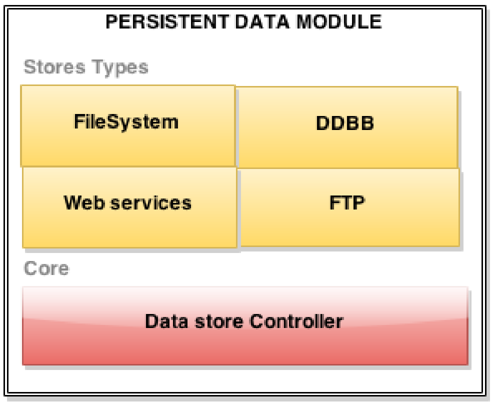


**Picture 16 Message module architecture.**

### 4.2.6. Persistent data module

This module is the responsible to store all the persistent information in the shell, configurations, results (reports and modified files) and logs.

The system will provide some default storage options, but developers could implement other ones using the factory pattern as explained before.

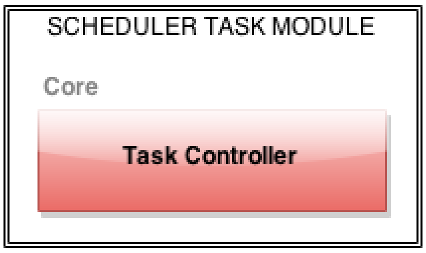


Picture 17 Persistent Data Module architecture.

The persistent data module can store validation results though a web service, this is especially useful for the **DIRECT infrastructure** (as required by the PREFORMA specifications) where the result can be stored in a remote key-value store server.

### 4.2.7. Task module

This module is responsible for triggering the execution of periodical conformance checks. The Scheduler task module acts as a daemon executing conformance checks at the desired time. The execution time is set inside the XML configuration using a standard Cron notation.



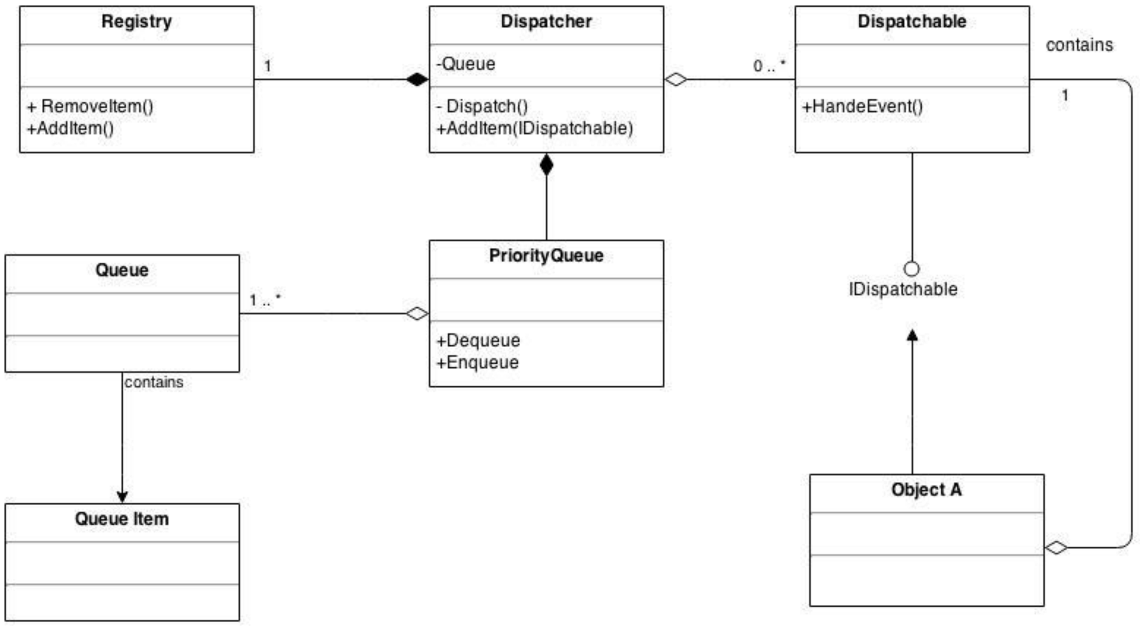
Picture 18 Scheduler Task Module architecture.

### 4.2.8 New modules

The sections above explain architecture of the core modules of the Shell. There will be situations where developers need to add new functionalities to the Shell. To enable the creation of new modules without losing decouplebility, the architecture provides an event-driven approach system using the proactor design pattern.

**Proactor pattern**

The Proactor Pattern is an asynchronous event handling pattern. This pattern dispatches multiple events from multiple threads to the corresponding event handlers. This allows the application to have multiple operations running simultaneously without requiring the application to have a corresponding number of threads. Besides, it creates a "low coupling" and "high cohesion" between objects or a layered architecture.



Picture 19 Proactor design pattern

- Dispatcher: the dispatcher dispatches the received events to the Dispatchables.
- Dispatchable: a Dispatchable object could receive events from the dispatcher.
- PriorityQueue: this queue contains the queued events (ordered in separate priority queues).
- Registry: contains the registered Dispatchable objects.
- IDipatchable: the interface to the Dispatchable objects.
- Object A: the Dispatchable object which will receive asynchronous events from the dispatcher.

The core architecture dispatch events when there is an interaction between modules Developers can extend the architecture registering events listeners when an event is dispatch.

**Module manager**

Using the OSGi (Open Service Gateway initiative) specifications as a base, the architecture provides a way to enable the loading of new modules developed by the community.

The OSGi (Open Service Gateway initiative) specification describes a modular system and a service platform for the Java programming language that implements a complete and dynamic component model.

Modules, coming in the form of bundles for deployment, can be started and stopped, without requiring a reboot; management of Java packages/classes is specified in great detail.. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

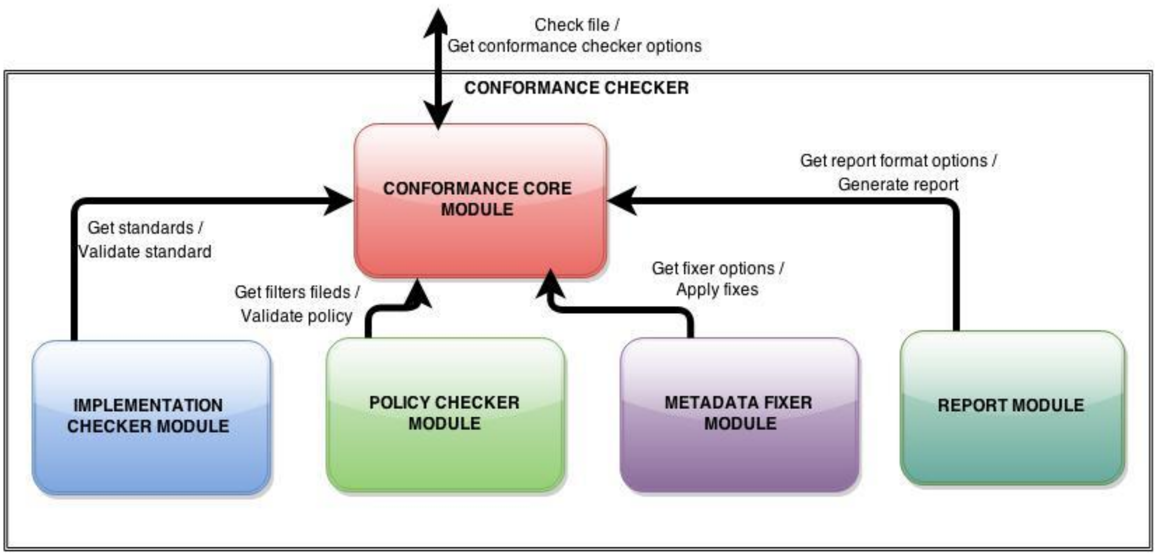
The OSGi specifications are now used in applications ranging from mobile phones to the open-source Eclipse IDE. Other application areas include automobiles, industrial automation, building automation, PDAs, grid computing, entertainment, fleet management and application servers.

## 4.3. Conformance Checker

Following our architecture principles the Conformance Checker has been designed in a modular way. The Conformance Checker modules are the following ones.

- Conformance Core Module: responsible of communication between the Shell and the Conformance Checker as well as all the interactions with the other modules.
- Implementation Checker Module: validate the Object Representation through standards.
- Policy Checker Module: validate the Object Representation through a defined criteria acceptance.
- Metadata Fixer Module: Modify the Object Representation metadata with preservation purpose.
- Report Module: Inform about Object Data structure and metadata information, validation results and fixes applied.

In the following sections all these modules functions and architectures are explained in detail with the interaction between the other modules.



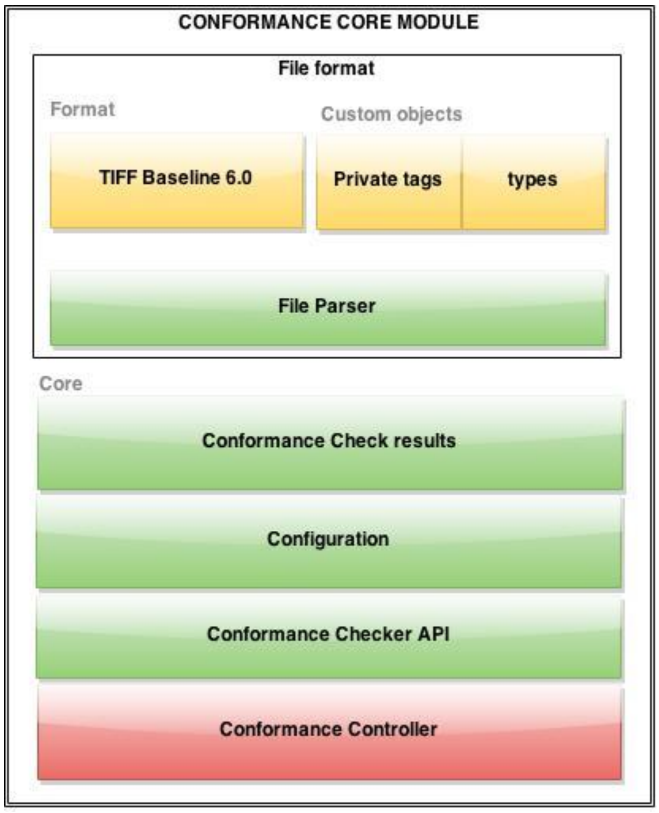
Picture 20 Conformance Checker modules and interactions.

### 4.3.1. Conformance Core Module

The conformance Core Module is the main component of the Conformance Checker. This module has the responsibility to coordinate all the other Conformance Checker modules and respond to the shell requests.

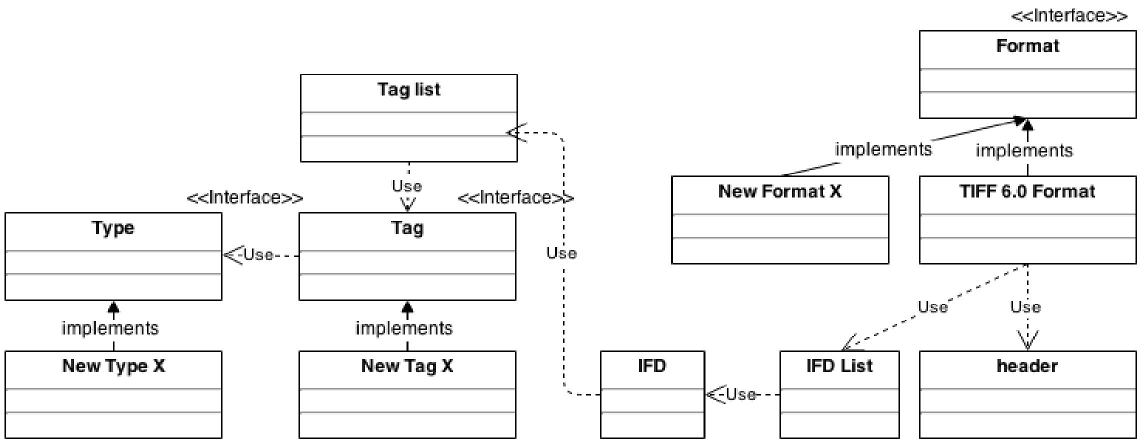
This component has to collect all the configuration options of the other modules as well as inform the Shell about how to identify the files that is able to check.

This module also collects all the results obtained during the conformance check process. The file format layer is able to read the file input and put the information in a format specific data structure call Object Representation. This structure is used for the other components in order to validate the file or modify it.



Picture 21 Conformance Core Module architecture.

In DPF manager an Object Representation implementation of TIFF is provided. This object stores all the information inside the TIFF file, the structure and the metadata information. The picture below shows the object implementation.



Picture 22 TIFF file format object structure.

The most relevant parts of the file format are the ability to add new tags and types. TIFF 6.0 baseline definitions include this possibility so our architecture let developers create new types and tags when it's required, for example if a new tiff standards include new ones or if they need to include new private tags or a new extension type is defined.

There are some tags in the TIFF format that contain complex information. This section explains this content and describes how to handle with complex information inside the TIFF format.

**Exif**

The "ISO 12234-2 Electronic still - picture imaging - Removable memory Part 2:TIFF/EP image data format" define how to include Exif (Exchangeable image file format) information inside a tiff file. This information is structured inside an IFD structure and the content is described inside a new set of tags. Too handle with this information is extremely easy because we can only have to implement this new tags and the TIFF format reader will be able to read this information.

**XMP**

Out of the baseline 6.0 tags definition there is an extended tag to store XMP information.

| Code | Name | Description | Source tag | Type |
|---|---|---|---|---|
| **700** | XMP | XML packet containing XMP metadata | Extended | BYTE |

This information is stored inside the TIFF using and XML format, using the default XML Java JDK libraries we are able to read the content. There is no need to include the Java framework provided by Adobe to work with XMP content.

**IPTC**

The "ISO 12234-2 Electronic still - picture imaging - Removable memory Part 2: TIFF/EP image data format" defines how to include IPTC (International Press Telecommunications Council-Newspaper Association of America) metadata inside a tiff file using the following tag.

| Code | Name | Description | Source tag | Type |
|---|---|---|---|---|
| **33723** | ITCP/NAA | IPTC-NAA metadata. | TIFF/EP | UNDEFINED |

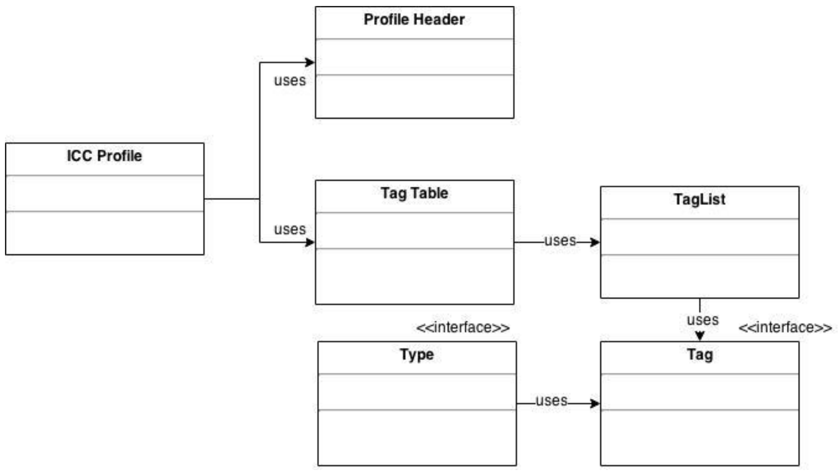
The information inside the ITCP tag is encoded using an XML format so the Java Libraries included in de JDK are enough to read the content.

**ICC Profile**

The ISO "ISO 12234-2 Electronic still - picture imaging - Removable memory Part 2: TIFF/EP image data format" and the "ISO 12639:2004 Graphic technology -- Prepress digital data exchange -- Tag image file format for image technology (TIFF/IT)" define a tag to embedded ICC colour profiles.

| Code | Name | Description | Source tag | Type |
|---|---|---|---|---|
| **34675** | ColorProfile | ICC profile data. | TIFF/EP and TIFF/IT | UNDEFINED |

The ICC Colour Profile follows a well-defined structure described in the "ICC.1:2010 Specifications Image technology colour management – Architecture, profile format, and data structure" currently version 4.3.



Picture 23 ICC Profile object structure

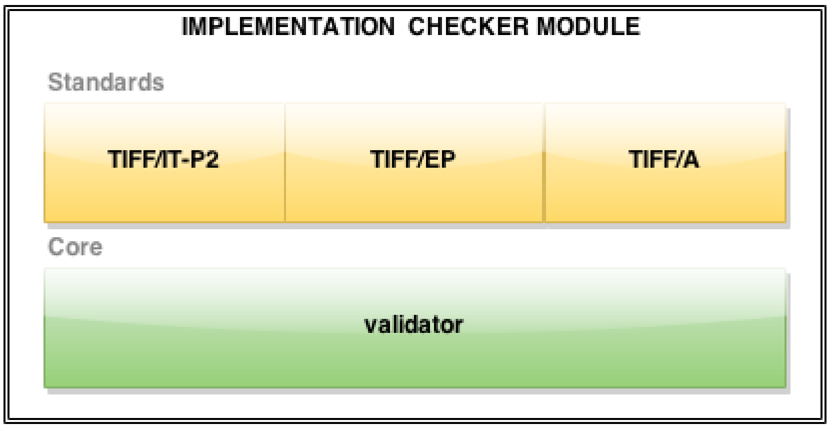
Using the ICC profile object structure we will be able to work with ICC profiles embedded inside the TIFF format.

All the information collected in these tags is send to the Metadata Fixer Module. The Metadata Fixer module verify the integrity of this data and make auto fixes over it if there are errors. Then apply the metadata changes required in the XML configuration file.

The reporter module includes all the metadata collected and fixed inside de XML report structure. All this metadata could be used to fill metadata standards structures like PREMIS or METS.

### 4.3.2. Implementation Checker Module

The Implementation Checker Module uses the Object Representation data structure that represents the input file, and validates this structure against a standardised format definition such as TIFF/IT or TIFF/EP.



Picture 24 Implementation Checker Module architecture

Three implementations are built in the DPF Manager application, the TIFF/IT and TIFF/EP matching with the PREFORMA specifications in addition to baseline specifications.

TIFF baseline 6.0 defines a group of sub classes based on the type of image stored in TIFF the following table shows the classes that the Implementation Checker Module can validate.

| Image type | TIFF Baseline |
|---|---|
| **Bilevel Images** | TIFF Class B |
| **Grayscale Images** | TIFF Class G |
| **Palette-Color Images** | TIFF Class P |
| **RGB Full Color Images** | TIFF Class R |
| **YCbCr Images** | TIFF Class Y |

Table 1 TIFF file types in the baseline 6.0.

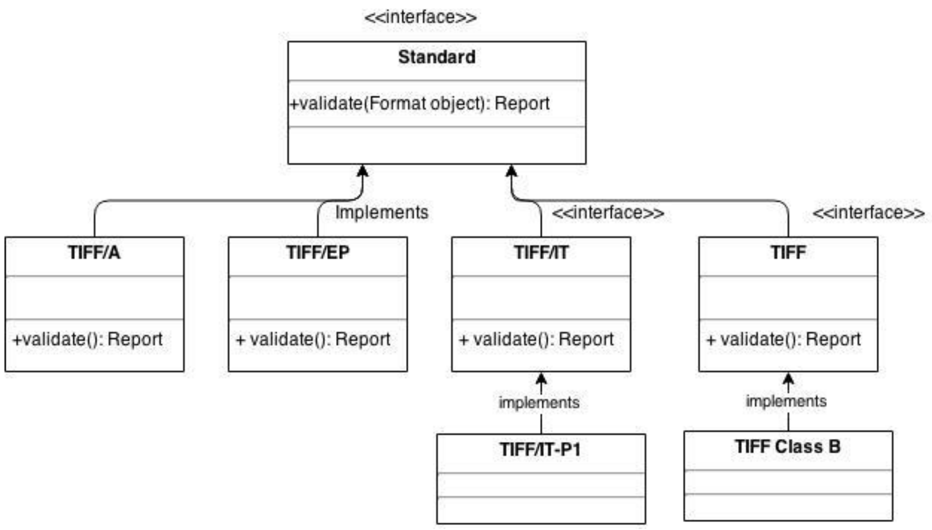
The TIFF/IT has two profiles TIFF/IT-P1 and TIFF/IT-P2 defined in the ISO 12639:2004. this two profiles and all images file types are taken into account in order to identify the file as the standard propose in Conformance identification section 2.6.

| Image type | TIFF/IT | TIFF/IT-P1 | TIFF/IT-P2 |
|---|---|---|---|
| **Colour continuous-tone picture image data (CT)** | TIFF/IT-CT | TIFF/IT-CT/P1 | TIFF/IT-CT/P2 |
| **Colour line-art image data (LW)** | TIFF/IT-LW | TIFF/IT-LW/P1 | TIFF/IT-LW/P2 |
| **High-resolution continuous-tone image data (HC)** | TIFF/IT-HC | TIFF/IT-HC/P1 | TIFF/IT-HC/P2 |
| **Monochrome continuous-tone picture image data (MP)** | TIFF/IT-MP | TIFF/IT-MP/P1 | TIFF/IT-MP/P2 |
| **Binary picture image data (BP)** | TIFF/IT-BP | TIFF/IT-BP/P1 | TIFF/IT-BP/P2 |
| **Binary line-art image data (BL)** | TIFF/IT-BL | TIFF/IT-BL/P1 | TIFF/IT-BL/P2 |
| **Screened data image data (SD)** | TIFF/IT-SD | TIFF/IT-SD/P1 | TIFF/IT-SD/P2 |
| **Final page data (FP)** | TIFF/IT-FP | TIFF/IT-FP/P1 | TIFF/IT-FP/P2 |

Table 2 TIFF/IT file types defined in the ISO 12639:2004.

Although it is not a PREFORMA requirement another implementation is proposed in the architecture. In our in-depth study of the TIFF standard, we realized that being ISO compliant is not enough to ensure the long preservation of a TIFF file. There are image representation structures and tags in the TIFF that are valid according to the standard but are not suitable for long term preservation.

As part of our proposal for the PREFORMA project we have defined a new TIFF subtype that is aimed at long term digital preservation.
The TIFF/A specification defines a valid file structure and classifies all the TIFF tags in 3 categories: mandatory, optional and forbidden.

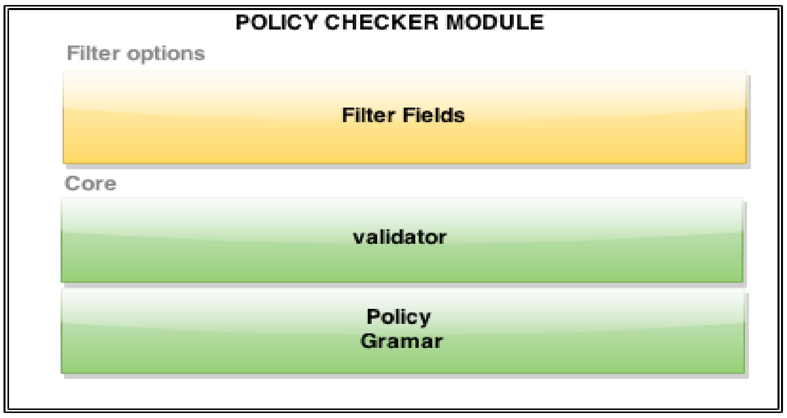


Picture 25 Standard Class structure with two sub-standards implementations

For each standard and sub-standard the corresponding class is provided. Each implementation develops the corresponding validation method taking the TIFF format object as an input.

### 4.3.3. Policy checker

The policy checker allows memory institutions to define a set of acceptance criteria based on their internal policies.



Picture 26 Policy checker module architecture

In order to define these policies the architecture provides a Policy grammar based on logical operators. This grammar is defined using an XML structure inside de configuration file and parsed to an Object Data. Then, the Policy rules can be validated against the TIFF format object representation.

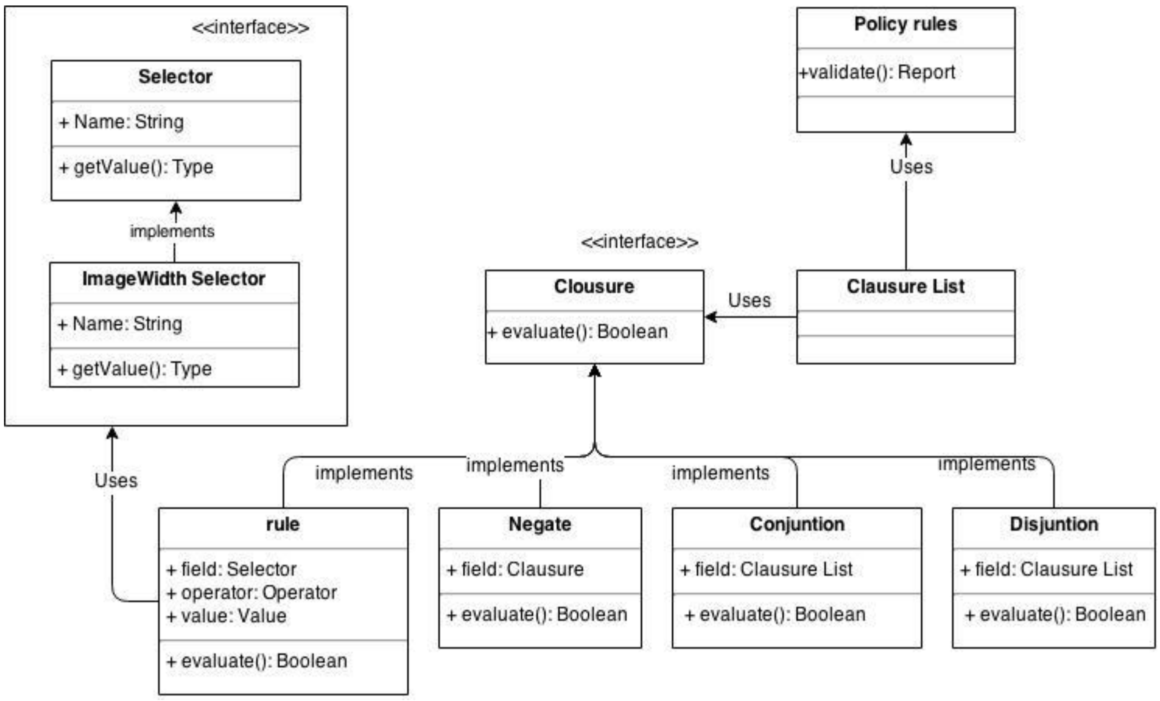
The valid logical operators are the conjunction (and), disjunction (or) and the negate operator (not). These operators can be combined with rules. A rule contains a selector that extracts the value from the Format Object representation, a comparison operator and the value to compare.

The selectors can be simple operations like get the width or height of an image, get colour profile or require a complex calculation like image processing algorithms.

These Object structure combinations provide a flexible way to define acceptance criteria suitable for different organizations.

The following pictures show the Object structure with all the elements required. Developers can easy implement new selectors and create new rules to define specific policies. This grammar enables the creation of simple rules as well as complex logical predicates.

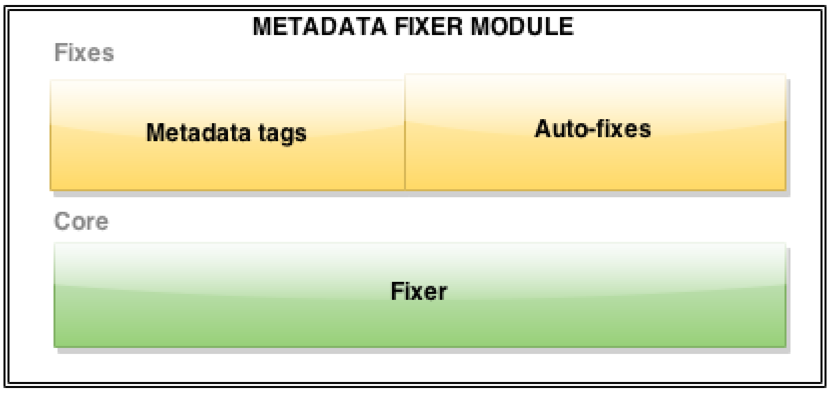
The policy rules object that contains the closures list has to call the validate method, that is propagated over the structure and rules, to generate a report with all the rules passed or failed.



Picture 27 Policy rules object structure.

### 4.3.4. Metadata fixer

The metadata fixer will perform simple changes to the metadata in the file to make it compliant with the standard/s defined in the implementation checker, or with the internal acceptance criteria of each memory institution.



**Picture 28 Metadata Fixer Module architecture**

The most common operation that memory institutions perform with the metadata is to add, remove and normalise information. But we wanted to provide a more flexible system to modify metadata. So we designed the auto-fixes concept. An auto-fix is a series of changes in the Format Object representation. These changes can be as simple as a group of metadata changes in order to remove private data in the TIFF or more complex like making a file ISO compliant with some standard.

Users can choose witch metadata they want to add and/or remove, and witch auto-fixes they want to apply.

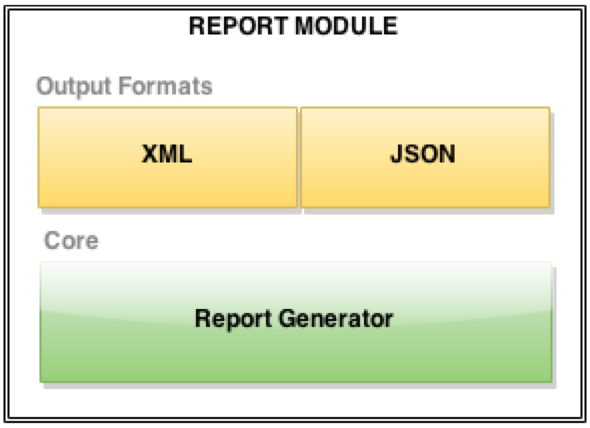
In addition they can choose if they want to override the original file or create a new one with the changes.

When the changes are applied the Format Object representation is written as a TIFF file.

### 4.3.5. Report Module

The Report Module is the responsible to parse the Report object generated by the Implementation Checker Module, the Policy Checker Module and the Metadata Fixer module, and aggregate this information in a single XML or JSON object. The file structure and list of tags are also added to this object.

In addition this component also serializes the Format object representation. This information could be used to fill in the Descriptive Information from the Archival Information Package in the OASI model.



Picture 29 Report Module Architecture

# 5. Data Flows

## 5.1. Introduction

This section describes the flow of data from/to the user, the shell, and the different components of the Conformance Checker .

To make it easier to understand, we have described the data flows for three different processes:

1. XML configuration validation and service discovery process: when the Shell receives a conformance check request with the configuration XML file, this file has to be validated. So, the Conformance Core Module queries all its components (implementation checker, policy checker, metadata fixer and reporter) about their capabilities (what they can do) in order to validate the XML configuration.

2. Conformance checking process: where one or multiple files are checked, fixed and the user receives a report with the result.

3. Historical data retrieval: where the users can see all the files that the shell has validated and fixed until now, together with the reports and all the information associated with each file. This can be especially useful when the shell is configured to do periodical checks on an archive, so the archive maintainers can always see the results of previous checks.

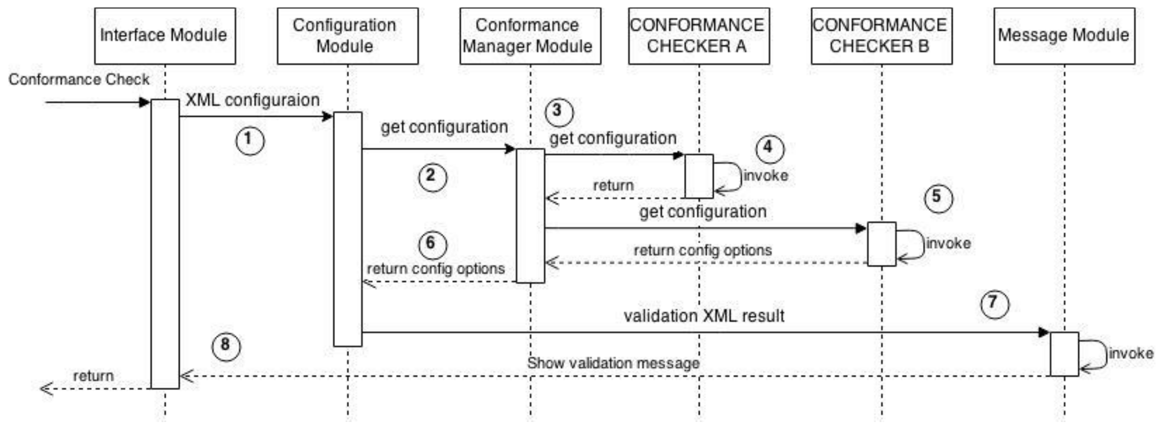
## 5.2. XML configuration validation and service discovery

The Shell has no hard-coded knowledge of the capabilities of the Conformance Checker.

When designing a modular system, it's good practice to make each module responsible of informing the other modules of what it can do. If we hard-coded this information into the shell, it would not be possible to update or replace any of the conformance checking modules (implementation checker, policy checker, etc..) without having to update the shell as well, which would defeat the purpose of having a modular system.

To get that information, the Shell has to query the Conformance Checker, which in turn queries all its individual components to get that information and pass it back onto the Shell. As soon as this information is available we can start validating the input XML file configuration.

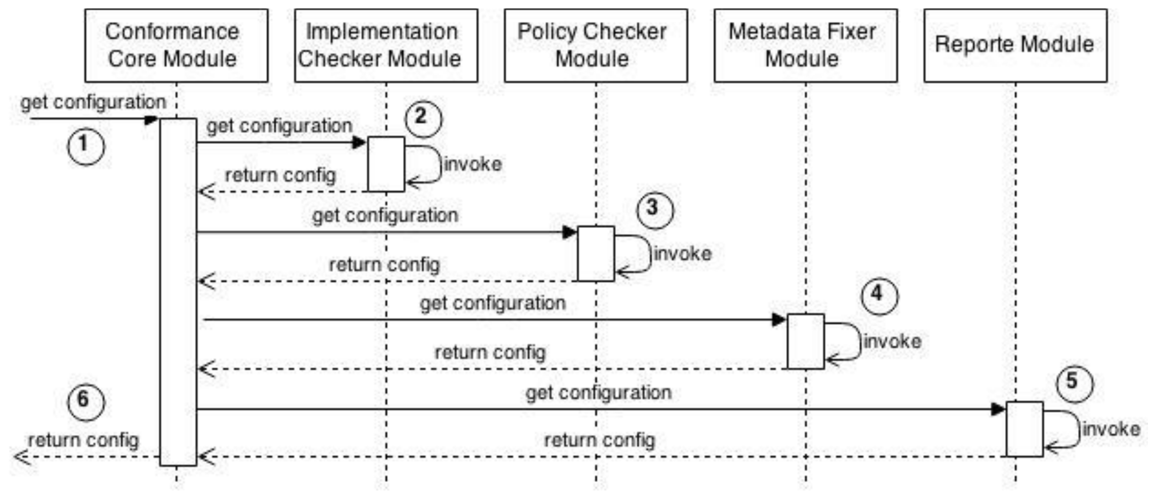
The following sequence diagram show the modules interaction inside the Shell.



Picture 30 validate XML sequence diagram

| | |
|---|---|
| 1 | When the Interface receives an XML configuration file it is sent to the configuration module in order to validate the file. |
| 2 | The configuration module asks the conformance manager module about the configuration. |
| 3 | The conformance manager module checks the configuration file. In this case the configuration file includes a server connection. Then the module has to configure the connection to the remote Conformance Checker B with the correct protocol (HTTP connection) before asking the configuration. |
| 4 | The conformance manager module asks the configuration to a build in Conformance Check A. |
| 5 | The conformance manager module asks the configuration for the Conformance Check B in a remote Shell in server mode, using the HTTP protocol. |
| 6 | All the configurations are returned to the Configuration module. Now it is able to validate the file. |
| 7 | Send the validation result to the Message module. This module returns the result to the interface. |
| 8 | The interface shows the validation result if it is needed. |

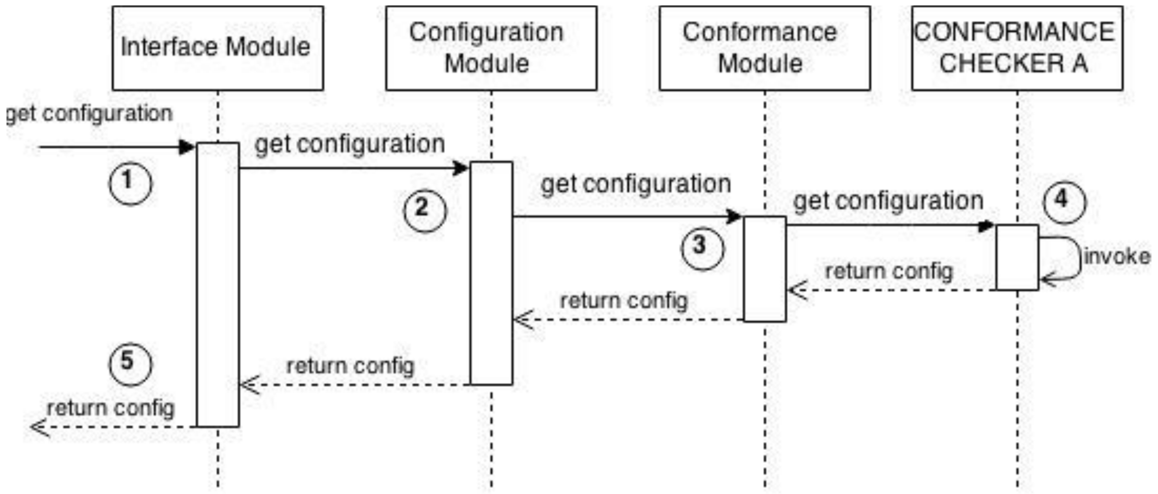
The following sequence diagram show the modules interaction inside the CONFORMANCE CHECKER A witch is a build-in conformance chequer.



Picture 31 get configuration sequence diagram

| | |
|---|---|
| 1 | The conformance core module receives a request to get the configuration options. |
| 2 | Ask the implementation checker module about which standards and sub standards is able to validate. |
| 3 | Ask the policy checker module about which policy rules is able to validate. |
| 4 | Ask the metadata fixer module about with metadata is able to add or remove inside the file and with automatic fixes are implemented in this module. |
| 5 | Ask the reporter module about which output format is able to report. |
| 6 | When all the configurations are returned to the Conformance core module adds the information about this conformance checker as well as the way to detect the files appropriate for this conformance. then, return all the configuration options |

The following sequence diagram show the modules interaction inside the CONFORMANCE CHECKER B which is a Conformance Checker in a remote shell in server mode.



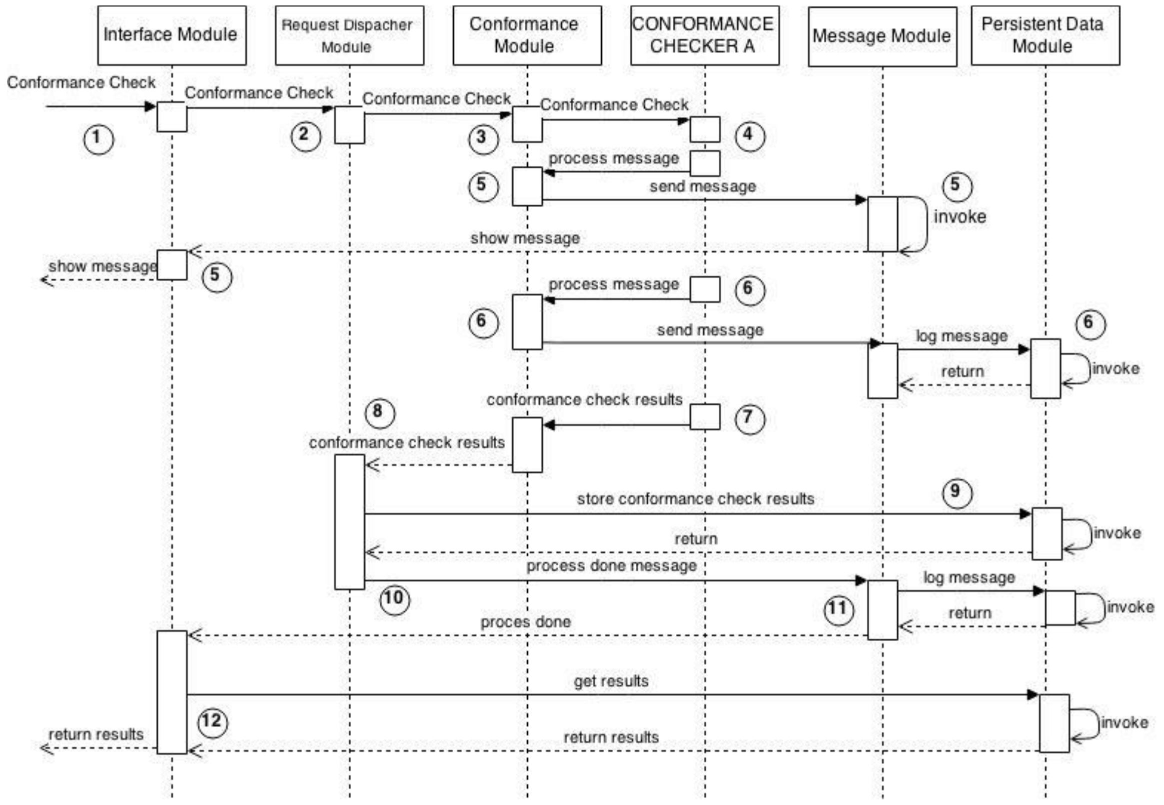
Picture 32 get configuration remote to a remote shell in server mode sequence diagram

| | |
|---|---|
| 1 | The Interface, in this case a server interface, receives a get configuration HTTP request and propagates the demand to the Configuration module. |
| 2 | The configuration module asks the Conformance module about the configuration. |
| 3 | The conformance module check the configuration for all the Conformance Checkers attached. |
| 4 | The conformance module asks the configuration to a build in Conformance Check A. this process has been explained in the previous sequence diagram. |
| 5 | The interface returns the configuration. |

## 5.3. Conformance checking

The following section explains the conformance checker process with a single file check and shows different interactions with the modules that take part in the validation process.

The process of validating the configuration file and the service exploration are omitted in the diagram because they were explained in the section before. In addition, some recurrent interactions has also been simplified to a single interaction. The first diagram shows the interactions inside the Shell.

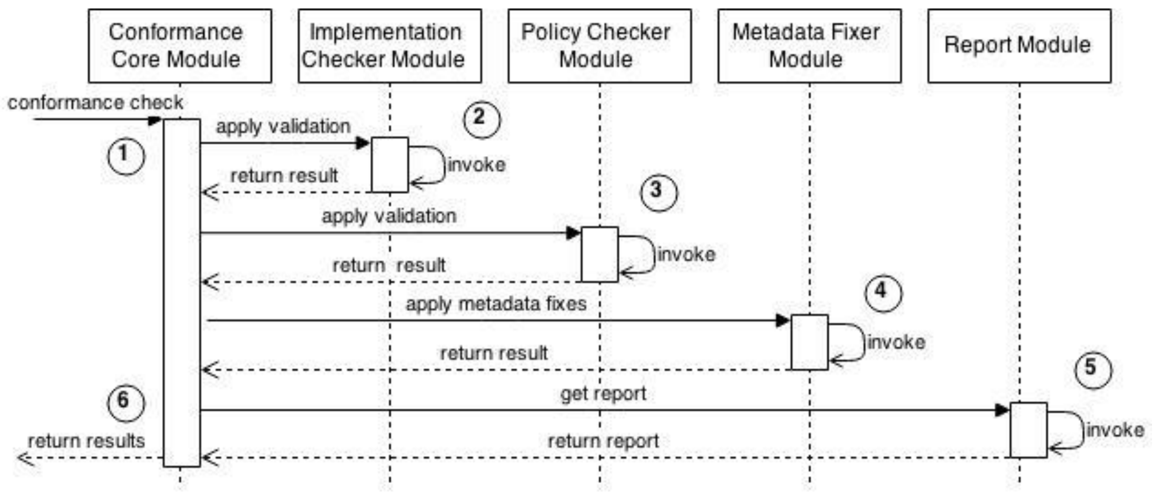


Picture 33 conformance check sequence diagram

| | |
|---|---|
| 1 | The interface receives the input source and a configuration file from the user. In the configuration file, the users specify which standards they want to validate against, which policies from the policy checker they want to use, what fixes will be performed on the files, and what format they want the report in. This information is sent to the request dispatcher module |
| 2 | The request dispatcher module processes the input source and extracts the files to the ReduceMap object. The file ReduceMap object and the configuration are sent to the conformance manager module. |
| 3 | The Conformance manager module apply the reduction with the identification function in order to detect the files appropriate for the CONFORMANCE CHECKER A |
| 4 | One file is send to the CONFORMANCE CHECKER A |
| 5 | During the process the conformance checker produce a message that is received by the conformance manager module. Send to the message module and distributed to the interface module in order to show the message. This interaction is repeated during all file validation process to inform the user. |

|  |  |
|---|---|
| 6 | The conformance checker produces a message that is received by the conformance manager module send to the message module and distributed to the persistent data module in order to log the message. This interaction is repeated during all file validation process to log all the actions. |
| 7 | The conformance checker returns the validation results to the conformance manager that returns the request dispatcher module. |
| 8 | The request dispatcher module generates the user readable global report and sends the results to the persistent data manager. |
| 9 | The persistent data manager store the information taking into account the configuration file |
| 10 | Then a message to inform about the process end is send to the message module |
| 11 | The message module logs the action using the persistent data module and sends the message to the interface. |
| 12 | The interface get the results from the persistent data module |

The following section show the sequence diagram inside the Conformance Checker



Picture 34 Conformance check sequence diagram

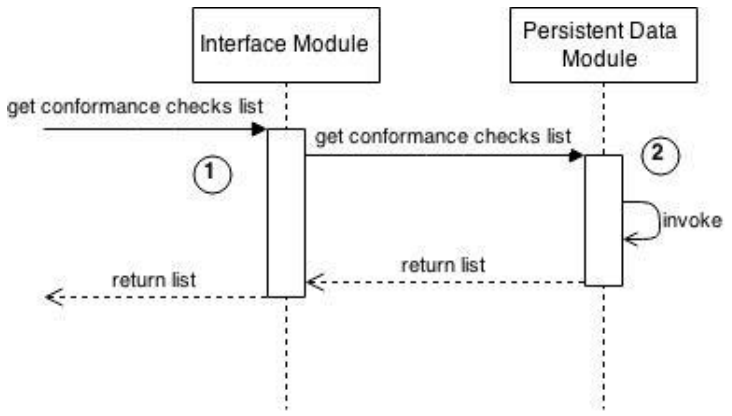
|  |  |
|---|---|
| 1 | The conformance core module receives the file and the options parameters. Then, parsers the file in a specific format structure object. This structure is used by the other modules. |
| 2 | The conformance core module calls the implementation checker module with the options received from the user (which standard to validate against), and the format |

| | |
|---|---|
| | structure object. The implementation checker returns a list of checks, indicating which ones have passed and which ones have failed.<br><br>The implementation checker can also return additional information for each failed check: why it failed, and what can be done to solve the issue. |
| 3 | The conformance core module calls the policy checker with the options received from the user (which policies should be checked), and the file structure and metadata. The policy checker returns a list of performed checks, indicating which ones have passed and which ones have failed. |
| 4 | The conformance core module calls the metadata fixer with the options received from the user (which may include a list of tags to add and a list of tags to remove), the file structure and metadata, and the results of the implementation checker and policy checker.<br><br>The metadata fixer will try to fix the errors found by the implementation and policy checker, and will also add and/or remove tags depending on the options defined by the user.<br><br>The result of the metadata fixer will be a list of performed and failed fixes and the location of the duplicate file in case the user has decided not to apply the fixes to the original one. |
| 5 | The conformance core module calls the reporter with the options received from the user (report formats and other options), and also the results of the implementation checker, policy checker and metadata fixer. |
| 6 | The conformance core module returns the results. |

## 5.4. Historical data

Users may need to see what files have been checked up till now, and what the result was. As the Shell can store the result of each transaction in the data store, this information can be easily retrieved in the future.

In this situation, there is no need to invoke the conformance checker. All the information we need is in the data store.



Picture 35 get validation history sequence diagram

| | |
|---|---|
| 1 | The Interface Module receives a request to list the checks that have been performed until now and what the results were. The interface asks this information to the Persistent Data module. |
| 2 | The Persistent Data Module returns the information to the user using the Interface Module. |

# 6. Data design and structure

## 6.1. Introduction

This section describes in detail the structure of the different types of data that the modules in the system can accept (data inputs) and produce (data outputs).

Each one of the Conformance Checker modules (implementation checker, policy checker, reporter and metadata fixer) needs some configuration information (telling the module what to do), and can produce two types of information: what the module is capable of doing (response to a service discovery request), and an interim report with the results of the module output.

## 6.2. Conformance checker

### Module capabilities

When queried about its capabilities, the Conformance Checker first gets this information from each one of the modules (implementation and policy checker, metadata fixer and reporter). It then adds some information of its own, like name, author, the type of files it can check, and how to detect these files:

```
<conformanceCheckerOptions>
    <name>TIFF/A conformance checker</name>
    <author>John Doe</author>
    <version>2.45b</version>
    <company>Easy Innova</company>
    <media_type>image/tiff</media_type>
    <extensions>
        <extension>tiff</extension>
        <extension>tif</extension>
    </extensions>
    <magicNumbers>
        <magicNumber>
            <offset>0</offset>
            <signature>\x49\x49\x2A\x00</signature>
        </magicNumber>
        <magicNumber>
            <offset>0</offset>
            <signature>\x4D\x4D\x00\x2A</signature>
        </magicNumber>
    </magicNumbers>

    <implementationCheckerOptions>...</implementationCheckerOptions>
    <policyCheckerOptions>...</policyCheckerOptions>
    <metadataFixerOptions>...</metadataFixerOptions>
    <reporterOptions>...</reporterOptions>

</conformanceCheckerOptions>
```

As one can see, the conformance checker aggregates the information obtained from the modules into a single response.

#### Configuration

When asked to check a tiff file, the conformance checker needs the configuration parameters for each one of the modules, so the configuration for the conformance checker is an aggregation of the configuration for each individual module:

```
<conformanceCheckerConfiguration>


<implementationCheckerConfiguration>...</implementationCheckerConfiguration>
    <policyCheckerConfiguration>...</policyCheckerConfiguration>
    <metadataFixerConfiguration>...</metadataFixeConfigurationr>
    <reporterConfiguration>...</reporterConfiguration>

</conformanceCheckerConfiguration>
```

#### Output

As the reporter is responsible for producing the final report, the conformance checker passes this information directly to the shell, without adding or changing anything.

### 6.2.1. Implementation checker

#### Module capabilities

When queried about its capabilities, the implementation checker responds with a list of the standards it can validate against:

```
<implementationCheckerOptions>
    <standards>
        <standard>
            <name>TIFF/A<name>
            <description>TIFF for long term digital
preservation<description>
        </standard>
        <standard>
            <name>TIFF/IT</name>
            <description>Standard for the exchange of digital adverts and
complete pages</description>
            <standards>
                <standard>
                    <name>TIFF/IT P1<name>
                    <description>TIFF/IT Profile 1<description>
                </standard>
                <standard>
                    <name>TIFF/IT P2</name>
                    <description>TIFF/IT Profile 2</description>
                </standard>
            </standards>
        </standard>
    </standards>
</implementationCheckerOptions>
```

For each standard a short description is provided. In case a standard has sub-standards (e.g. TIFF/IT has TIFF/IT-P1 and TIFF/IT-P2), the sub-standard are also included.

### Configuration

When asked to check a TIFF file, the implementation checker needs to know which standard it should validate against. This information is passed to the implementation checker by the conformance checker:

```
<implementationCheckerConfiguration>
    <standards>
        <standard>TIFF/A</standard>
    </standards>
</implementationCheckerConfiguration>
```

### Output

Once the implementation checker finalizes checking a file, it produces an interim report with the results:

```
<implementationCheckerOutput>
    <results>
        <result>
            <level>critical</level>
            <tag>265</tag>
            <msg>Tag 265 - CellLength is forbidden in TIFF/A. Dithering is a
not allowed concept of tonal representation in an image.</msg>
        </result>
        <result>
            <level>info</level>
            <tag>278</tag>
            <msg>Tag 278 - RowsPerStrip tag is present and has a valid
value.</msg>
        </result>
    </results>
</implementationCheckerOutput>
```

Each check performed by the implementation checker can result in either a pass or fail. This information is found in the <level> tag and can have these values:

| Level | Description |
|---|---|
| critical | Check failed: a mandatory tag is missing, a forbidden tag is present, the TIFF file has an un-supported structure, the data is compressed, etc. The conformance process did not pass. |
| warning | Check passed, but action is **strongly** recommended. It is used for tags that are not mandatory but highly recommended. E.g. the tag 'ImageDescription' is not mandatory, but a brief embedded information about the image content is important and can be seen as an increasing factor for redundancy. |
| notice | Check passed, but action is recommended. It is used for tags that are not mandatory but recommended. E.g. the tag 'Copyright' is not mandatory, but digital rights are important and if possible this tag should be filled. |
| info | Check passed, no further action needed. |

### 6.2.2. Policy checker

#### Module capabilities

When queried about its capabilities, the policy checker responds with a list of tags and predefined functions that it can check, what type of values it can have (numbers, strings, etc.), an optional list of values it can accept, and a short description:

```
<policyCheckerOptions>
    <fields>
        <field>
            <name>ImageHeight</name>
            <type>integer</type>
            <description>Image height in pixels</description>
        </field>
        <field>
            <name>BitsPerSample</name>
            <type>integer</type>
            <values>16,32</values>
            <description>Number of bit per sample</description>
        </field>
    </fields>
</policyCheckerOptions>
```

This allows users to know what options they have to define their own policies.

#### Configuration

When asked to check a TIFF file, the policy checker needs to know which policies should be validated. This information is passed to the policy checker by the conformance checker:

```
<policyCheckerConfiguration>
    <policies>
        <policy>
            <field>
                <name>BitsPerSample</name>
                <operator>=</operator>
                <value>16</value>
            </field>
        </policy>
        <policy>
            <and>
                <field>
                    <name>ImageWidth</name>
                    <operator>=</operator>
                    <value>3500</value>
                </field>
                <field>
                    <name>ImageHeight</name>
                    <operator>=</operator>
                    <value>2500</value>
                </field>
            </and>
        </policy>
    </policies>
</policyCheckerConfiguration>
```

Each policy is defined by a name (either a tag or a predefined function), an operator and a value.

### Output

Once the policy checker finalizes checking a file, it produces an interim report with the results:

```
<policyCheckerOutput>
    <results>
        <result>
            <level>critical</level>
            <policy>ImageWidth >= 3500px and ImageHeight >= 2500px</policy>
            <msg>The  image  does  not  conform  to  this  policy.  ImageWidth  =
1500px and   ImageHeight = 2000px</msg>
        </result>
        <result>
            <level>info</level>
            <policy>BitsPerSample = 16</policy>
            <msg>The image BitsPerSample = 16</msg>
        </result>
    </results>
</policyCheckerOutput>
```

Each check can result in a pass or fail. This information is found in the <level> tag, that can have these values:

| Level | Description |
|---|---|
| critical | Check failed. The file does not conform to the policy. |
| info | Check passed, no action needed. |

### 6.2.3. Metadata fixer

#### Module capabilities

When queried about its capabilities, the metadata fixer responds with a list of tags that it can add (and its possible values) and a list of tags that it can remove from the file:

```
<metadataFixerOptions>
    <addTags>
        <tag>
            <name>Copyright</name>
            <id>33432</name>
            <type>string</type>
            <description>Copyright notice</description>
        </tag>
        <tag>
            <name>ImageDescription</name>
            <id>270</name>
            <type>string</type>
            <description>Short description of image content</description>
        </tag>
    </addTags>
    <removeTags>
        <tag>
            <name>DateTime</name>
            <id>306</name>
            <description>Date and Time of digital image creation
</description>
        </tag>
    </removeTags>
</metadataFixerOptions>
```

#### Configuration

When asked to fix a TIFF file, the metadata fixer needs to know:

1. If it should attempt to automatically fix errors or not.
2. If the fixes should be applied to the original file or a duplicate should be created.

3. What tags it should add to the file (and if the tag is already there, if it should be replaced or not)
4. What tags it should remove from the file.

This information is passed to the policy checker by the conformance checker:

```
<metadataFixerConfiguration>
    <enableErrorFixing>true</enableErrorFixing>
    <duplicateFile>true</duplicateFile>
    <duplicateFileName>image1_fixed.tiff</duplicateFileName>
    <addTags>
        <tag>
            <id>33432</id>
            <name>Copyright</name>
            <value>© Easy Innova 2015</value>
            <replace_if_exists>true</replace_if_exists>
        </tag>
    </addTags>
    <removeTags>
    </removeTags>
</metadataFixerConfiguration>
```

### Output

Once the metadata fixer finalizes, it produces an interim report with the results:

```
<metadataFixerOutput>
    <duplicateFileName>image1_fixed.tiff</duplicateFileName>
    <results>
        <result>
            <level>critical</level>
            <tag>33432</tag>
            <msg>Tag 33432 - Copyright could not be added.</msg>
        </result>
        <result>
            <level>info</level>
            <tag>306</tag>
            <msg>Tag 306 - DateTime was removed correctly.</msg>
        </result>

    </results>
</metadataFixerOutput>
```

The output gives information about whether the fix was applied successfully or not. This information is found in the <level> tag and can have these values:

| Level | Description |
|---|---|
| critical | Metadata fixer was not able to fix the issue. |
| info | Issue fixed. |

### 6.2.4 Reporter

#### Module capabilities

When queried about its capabilities, the reporter responds with a list formats it can produce, and an optional list of modes for each format:

```
<reporterOptions>
    <formats>
        <format>
            <type>xml</type>
        </format>
        <format>
            <type>html</type>
            <modes>
                <mode>expert</mode>
                <mode>novice</mode>
            </modes>
        </format>
        <format>
            <type>pdf</type>
            <modes>
                <mode>expert</mode>
                <mode>novice</mode>
            </modes>
        </format>
    </formats>
</reporterOptions>
```

#### Configuration

When asked to produce a report, the reporter needs to know what formats the user has requested. For some formats, different modes are available, e.g. the HTML and PDF reports can be tailored to experienced and novice users.

This information is passed to the reporter by the conformance checker:

```
<reporterConfiguration>
    <formats>
        <format>
            <type>xml</type>
        </format>
        <format>
            <type>html</type>
            <modes>
                <mode>expert</mode>
            </modes>
        </format>
    </formats>
</reporterConfiguration>
```

In this example we are asking the reporter to produce an XML report and a HTML report in expert mode.

## Output

When calling the reporter, the conformance checker gives it the interim reports produced by the implementation checker, policy checker and metadata fixer.

The job of the reported is to put the interim reports together into a final report, and to format the report according to the configuration provided.

An example of a final XML report is included below:

```
<report>

    <implementation_checker>
        <results>
            <result>
                <level>critical</level>
                <tag>265</tag>
                <msg>Tag 265 - CellLength is forbidden in TIFF/A. Dithering is a not
                     allowed concept of tonal representation in an image.</msg>
            </result>
            <result>
                <level>info</level>
                <tag>278</tag>
                <msg>Tag 278 - RowsPerStrip tag is present and has a valid value.</msg>
            </result>
        </results>
    </implementation_checker>

    <policy_checker>
        <results>
            <result>
                <level>critical</level>
                <policy>ImageWidth >= 3500px and ImageHeight >= 2500px</policy>
                <msg>The image does not conform to this policy. ImageWidth = 1500px and
                     ImageHeight = 2000px</msg>
            </result>
            <result>
                <level>info</level>
                <policy>BitsPerSample = 16</policy>
                <msg>The image BitsPerSample = 16</msg>
            </result>
        </results>
    </policy_checker>

    <metadataFixer>
        <duplicateFileName>image1_fixed.tiff</duplicateFileName>
        <results>
            <result>
                <level>critical</level>
                <tag>33432</tag>
                <msg>Tag 33432 - Copyright could not be added.</msg>
            </result>
            <result>
                <level>info</level>
                <tag>315</tag>
                <msg>Tag 315 - Artist was removed correctly.</msg>
            </result>
        </results>
    </metadataFixer>

</report>
```

## 6.3. Shell

### Module capabilities

When queried about its capabilities, the shell forwards the request to the conformance checker, and returns the result without modifying it.

If the shell is configured to work with multiple conformance checkers, it aggregates the data from each conformance checker and presents all the results at once:

```
<conformanceCheckersOptions>

    <conformanceCheckerOptions>
        <name>TIFF/A conformance checker</name>
        <author>John Smith</author>
        <version>2.45b</version>
        .....
    </conformanceCheckerOptions>

    <conformanceCheckerOptions>
        <name>TIFF/IT conformance checker</name>
        <author>Jane Doe</author>
        <version>1.01</version>
        .....
    </conformanceCheckerOptions>

</conformanceCheckersOptions>
```

### Configuration

The configuration is the same as for the conformance checker (previous point).

### Output

The output of the shell is the report as described in the output of the conformance checker, plus the files (if any) that have been modified by the metadata fixer.

When a user asks the shell to check multiple files, the shell calls the conformance checker once per file, so if we want to check 10 files, the shell will make 10 calls to the conformance checker.

In this situation, the shell will create a summary report. This report will display a summary of the results for each file, so users can easily see how the checking process went in a single screen, without having to open the reports for each individual file. This summary report is described in more detail in the Reporting section of the document.

# 7. Languages and frameworks

## 7.1. Introduction

Selecting the right programming language can have a significant impact on the chances of success of any given project. For us, the language selection process has taken into account the following criteria:

1. Good fit for the requirements of the PREFORMA project.
2. Popularity amongst developers in the open source community.
3. Easy Innova proficiency level.

After a thorough review of all the possible options (full details below), we have concluded that Java is the programming language that better matches the three criteria, and therefore is the best suited for this type of project.

## 7.2. Programming language selection criteria

Listed below are the criteria we have used to select Java as our preferred programming language for this project.

### 7.2.1. Good fit for the requirements of the PREFORMA project

Portability and platform independence: Thanks to the use of the JVM (Java Virtual Machine), any piece of code written in Java can be deployed and executed in almost any operating system (from Windows, OS X and Unix/Linux to Android, smart watches and other embedded devices).

Multiple interfaces: Unlike some other languages that were designed for specific purposes, Java is a language that allows developers to build software as diverse as command line clients, graphical user interfaces, and web platforms and services. This flexibility, together with the fact that it can be deployed to almost any environment, has been key to us selecting Java for this project.

Java is an object oriented language, which allows software to be designed in a modular way, with decoupled components and services. This makes it easier to scale and expand a well-designed architecture, and also makes it much easier to maintain.

Java is also a very mature programming language, having been around since the early 90's, and thanks to its popularity (see next point), it is very unlikely that it will fade into irrelevance any time soon.

To make it even more attractive, there is a huge ecosystem of well-tested open source libraries and services (some of them aimed at still image processing) that make it easier for developers to reuse existing solutions, and also make it more popular with the open source community.

## 7.2.2. Popularity amongst developers and the open source community

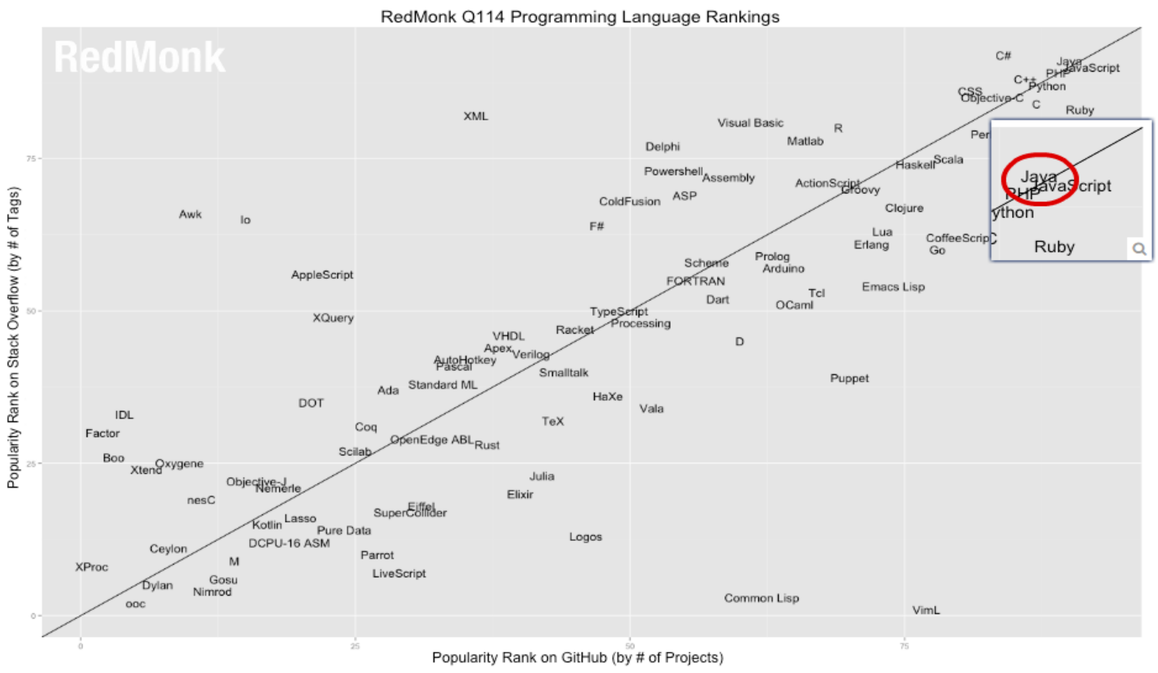
The PREFORMA requirements put a strong emphasis on the need to involve the open source community.

Whatever language is chosen, it needs to be popular not only amongst developers in general, but also in the open source community in particular, to ensure the widest adoption possible.

A good way to estimate the popularity of a programming language amongst developers is the popularity of the language in the Q&A website stackoverflow.com. StackOverflow is one of the most popular sites in the software development community.

When talking about open source, the reference site is github.com. Github is a web-based Git repository hosting system that is home to more than 5 million open source projects.

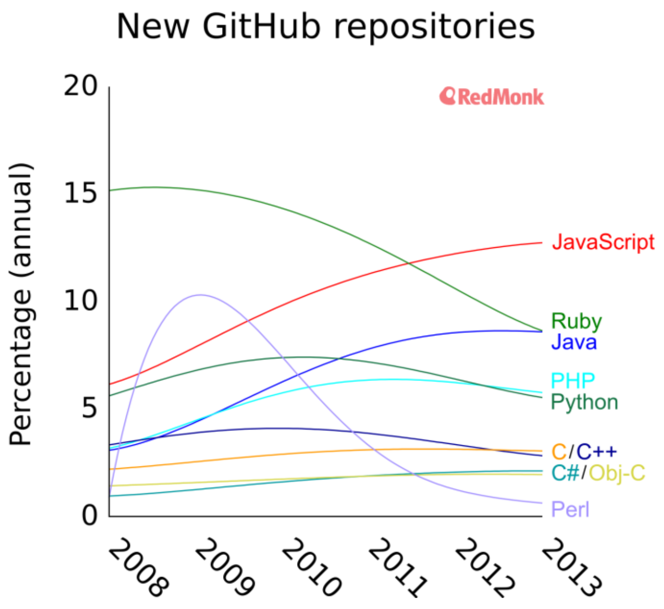
To determine the overall popularity of a programming language, the following chart cross-references the number of technical questions in StackOverflow and the number of open source Github projects by language. It was produced by the software development industry analysis firm RedMonk in May 2014.



As one can see, Java is the 2nd most popular language in both platforms (being C# the most popular in StackOverflow, and JavaScript in Github).

So in terms of popularity in the open source community (and the software development community in general), Java is one of the best languages one could choose.

The next chart (also produced by RedMonk in May 2013) shows how language popularity has evolved over time on Github, by looking at the number of new projects created each year for each language:



One of the key takeaways is that, aside from being the 2nd most popular language in Github, its popularity is growing rather than diminishing (unlike Ruby for instance). That fact gives us even more confidence that Java is a good choice for this project.

### 7.2.3. Proficiency level in supplier

Easy Innova has 10+ years of experience developing Java solutions for the private and public sector, and for different environments such as web and desktop applications.

In terms of digital preservation, Easy Innova is part of the consortium developing the DURAFILE European project (www.durafile.eu), also using Java. So we are confident we can deliver a solid solution for the PREFORMA project.

## 7.3. Programming language version

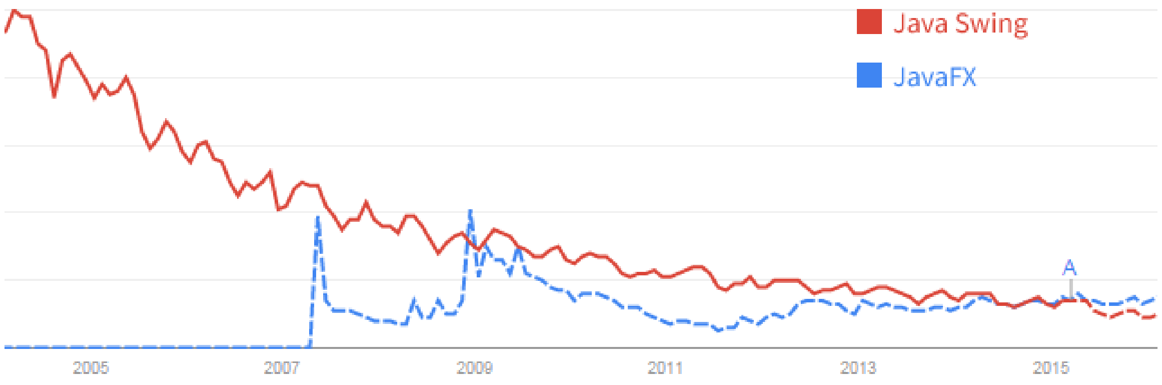
In terms of which Java version we would set as a minimum requirement, most of the Java functionality and components we are planning on using are pretty standard and compatible with Java 6 and upwards.

As the DPF Manager includes a Graphical User Interface (GUI), we have looked at all the graphical libraries that are available for Java: Swing, SWT and JavaFX.

SWT is licensed under EPL (Eclipse Public License), which is not compatible with the DPF Manager license (GPL3+ and MPL2+).

Swing is the legacy way of building Java visual interfaces. It is outdated and is being replaced by JavaFX.

The following chart shows the evolution of Google Search terms for both Java Swing and JavaFX, which can be used as a proxy to estimate the popularity of both libraries. As one can see, JavaFX did overtake Swing at the beginning of 2015, and the tendency (slashed lines after point A) is upwards for JavaFX and downwards for Swing.



For this reason we have decided to use JavaFX. The only drawback when comparing it with Swing is that JavaFX requires Java 7 or later.

Therefore the DPF Manager will require Java 7 or newer.

## 7.4. Use of third party libraries

### 7.4.1 Introduction

The Architecture proposed by EASY INNOVA has been designed to be developed from scratch. Aside from the 3rd party libraries explained in this section, we do not want to reuse any exiting code already developed for EASY INNOVA in other projects and for other purposes.

The goal of developing most of the DPF manager from scratch is:

- Avoiding compromising any of our design principles trying to fill already developed software in a new architecture.
- Developers do not have to learn other frameworks or libraries that will increase the DPF manager framework learning curve.
- No need to sublicense any existing software.
- The lack of innovation that could involve using already developed software.

We can overcome the lack of libraries in the core application working in the following points.

- **Domain expertise:** Library authors are usually experts in the domain covered by the library. We have deeply studied all the standards involved in the Conformance Checker , and we are working together with experts in image preservation from the University of Basel in order to develop the DPF Manager Framework.

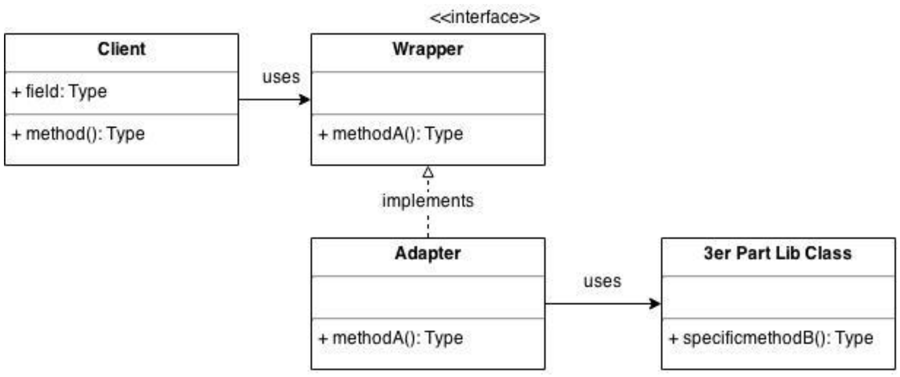
- **Stability:** 3rd party libraries are being used by other people as well as you, and in many cases, hundreds if not many thousands of developers worldwide. Most of the early problems have already been detected by others and fixed by authors. We hope that our developer's community will give us this stability in the future but meanwhile we will perform unitary test and use of integration platforms to counterbalance.

- **Financial impact:** We have evaluated the viability in cost and time in terms of developing our Conformance Checker from scratch to ensure that is completely possible.

There are some points in the architecture where the use of 3rd party libraries is mostly required. We have evaluated different frameworks and we have chosen the libraries taking into account; libraries already used successfully in other projects, widely used, stable, tested, active community, bugs fixed and well-documented.

In the Architecture design we also take into account the possibility to replace a currently used library for another one using the adapter pattern.

With the Adapter patter we could wrap the third party library class with and interface (Wrapper) that we expose to the developers. Developers will directly use the exposed interface and not have to directly call third party library methods. This provides a way to shield or break interfaces when a third party library changes something. To use another library you only have to implement a new Adapter.



Picture 36 Adapter pattern

In the following sections we explain where the libraries selected are going to be used as well as why there are used.

### 7.4.2. General libraries

All Java versions include a library known as JDK. This base library distributed under GPLv2+CE is suitable for most cases but have some missing functionalities.

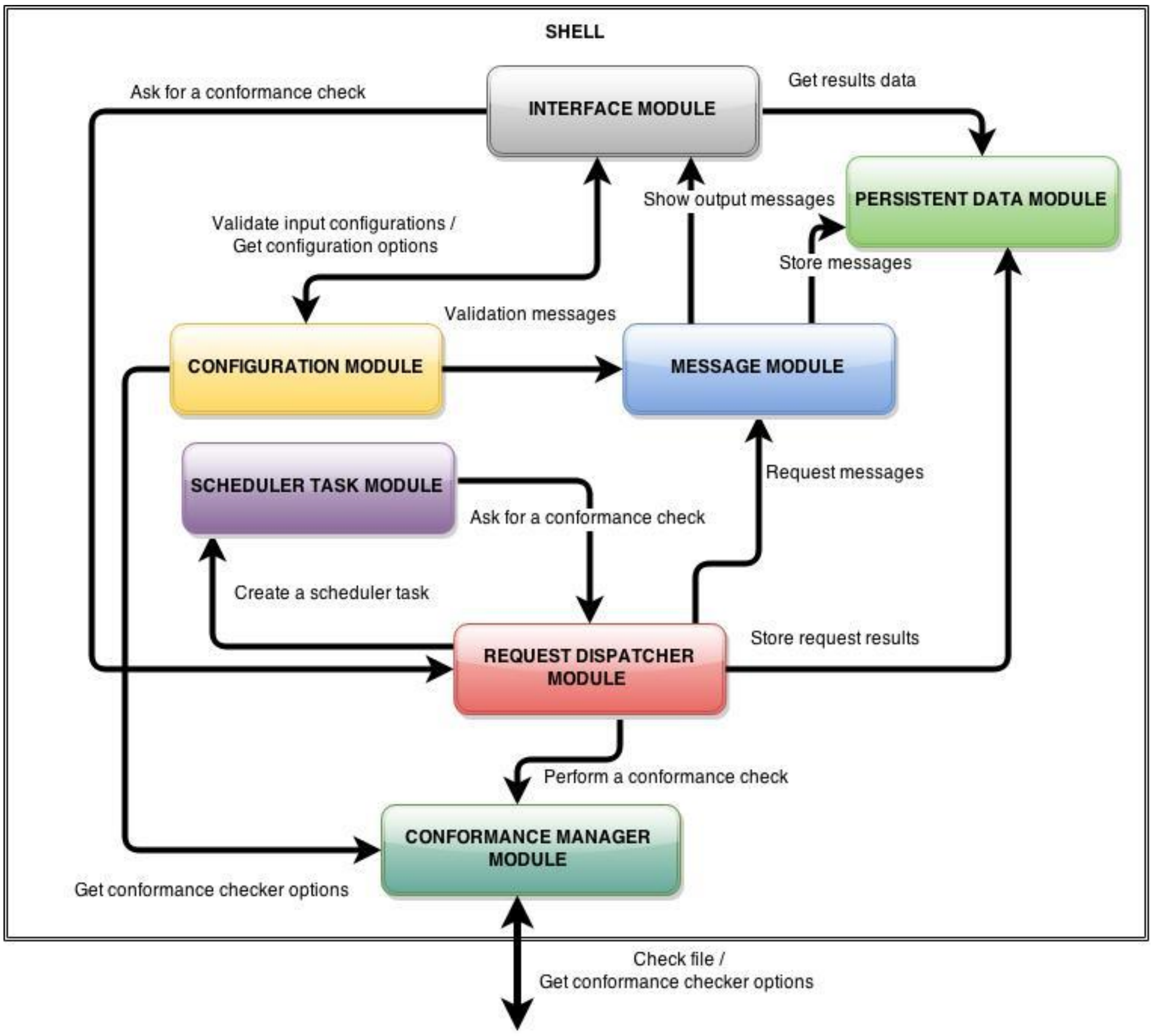
Apache commons is one of the most widely used libraries in Java. This library implements some of common missing functionalities in the JDK or extends the existing ones. Apache Common components do not have dependencies on other libraries, so that these components can be deployed easily. The Apache Common components that are going to be use are the following ones.

- **Commons CLI**: provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool. This component is suitable for the command line interface.
- **Common Compress**: defines an API for working with ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200, bzip2, 7z, arj, lzma, snappy, DEFLATE and Z files. This component will be used to process compressed input sources.
- **Common Configuration**: provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources. Will be used to set the general configuration parameters for the DPF manager.
- **Common Discovery**: The Discovery component is about discovering, or finding, implementations for pluggable interfaces. It provides facilities for instantiating classes in general, and for lifecycle management of singleton (factory) classes. This component discovered the new modules, objects added by developers to implement new functionalities**.**
- **Common Email**: Library for sending e-mail from Java. Used to inform about a scheduled conformance check action.
- **Common Launcher**: Cross platform Java application launcher.
- **Common logging**: Wrapper around a variety of logging API implementations used to log messages.
- **Common Validator**: Framework to define validators and validation rules in an xml file. Used to validate the XML input configuration and the XML report generate by the conformance checker.
- **Common VFS**: Virtual File System component for treating files, FTP, SMB, ZIP, WEBDAV and such like as a single logical file system. Used to support remote input file sources and remote store files.

All the Apache Common components are distributed over Apache commons 2.0 license that ensure the compatibility with DPF Manager license.

### 7.4.3. Shell 3rd party libraries

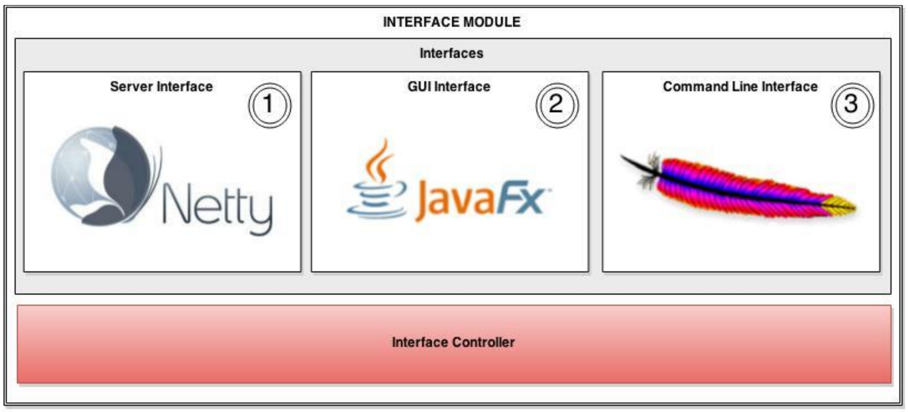
In the following section we describe where in the shell architecture we going to use 3rd party libraries and the purpose of this libraries.


SHELL
Ask for a conformance check
INTERFACE MODULE
Get results data
Show output messages
PERSISTENT DATA MODULE
Validate input configurations / Get configuration options
Store messages
Validation messages
CONFIGURATION MODULE
MESSAGE MODULE
SCHEDULER TASK MODULE
Request messages
Ask for a conformance check
Create a scheduler task
REQUEST DISPATCHER MODULE
Store request results
Perform a conformance check
CONFORMANCE MANAGER MODULE
Get conformance checker options
Check file / Get conformance checker options

Picture 37 Shell modular architecture

### The Interface Module

The Interface Module provides the access to the DPF manager functionalities. We understand that develop a server interface and a user graphical interface is completely out of PREFORMA project because the amount of time and money that could carry, as well as compromise the stability of the hole project. In other to provide the interfaces required by PREFORMA we choose the following libraries.



Picture 38 Interface Module libraries

1. **Netty**. Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server, and large file tranfer. Netty is a mature framework, widely used (see: http://netty.io/wiki/adopters.html) with an extensive documentation.

   We also evaluate the use of other frameworks like Jetty. Jetty is more appropriate when you want to provides a Web server with a javax.servlet container to deploy WAR files while Netty is more suitable for a true Client-Server application.

   In this module Netty implements the server communication.

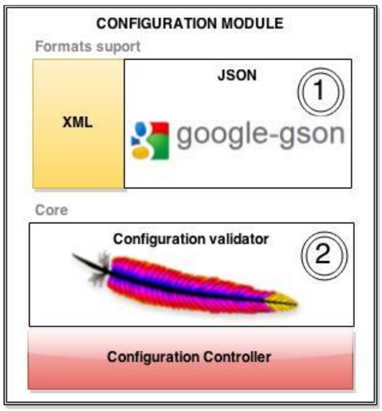
2. **Java FX**. Included in Java 7 JDK, JavaFX is a set of graphics and media packages that enable developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. With this framework we could create and attractive user interfaces for the standalone version.

3. **Commons Cli**. Included in Apache Commons the Cli component provides the tools necessary for parsing command line options to be interpret.

### Configuration Module

Java JDK 7 has the necessary tools to serialize XML files but not include tools to serialize JSON therefore we need a library to supply this lack.

1. **Google-gson**. One of the most used libraries to work with JSON objects in Java. Google-gson can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects.

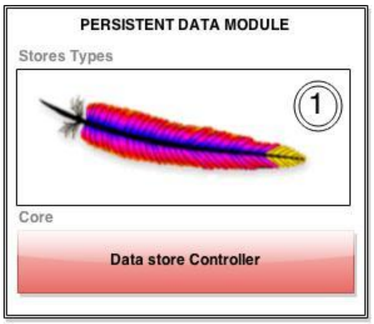


Picture 39 Configuration Module libraries

2. **Common validator**. Verifying the integrity of the input configuration files to ensure a successful file check is crucial. Although Java JDK7 provides a way to validate a XML through and XSD Schema, this library provides a set of utilities to validate different kind of data types.

**Persistent Data Module**

1. **Common logging**. The most famous library for logging in Java. The use of this library will ensure that we are able to store the logging persistent data in our file system.



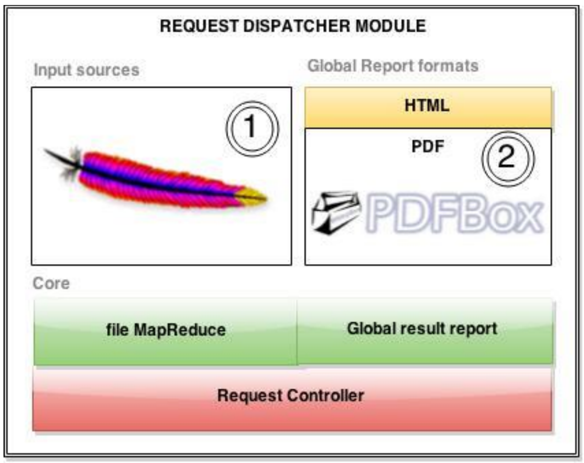
Picture 40 Persistent Data Module

**Request Dispatcher module**

This module is responsible to read the input source. We need to read as most input sources as possible to give the DPF manager flexibility to be suitable in different environments.

1. **Common VFS** and **Common Compress**. The common VFS give access to a large number of remote file systems and sources (see: http://commons.apache.org/proper/commons-vfs/filesystems.html) and with Common Compress we could process the most common compressed files formats.

2. **PDFBox**. Open source Java tool for working with PDF documents. This library allows creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents. Used to generate a global report with templates.

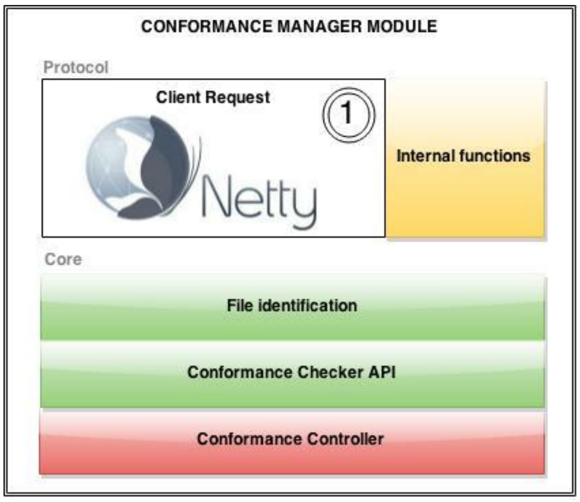


Picture 41 Request dispatcher module library

**Conformance Manager Module**

The conformance manager module is able to interact with the build-in Conformance Checkers as well as remote Conformance Checkers in a DPF manager server.

1. **Netty**. In this module Netty is used to implement the client communication with the server implementation.



Picture 42 Conformance manager module library

### 7.4.4. Conformance Checker without dependencies

The Conformance Checker internal architecture is structured in modules. All of them free of 3rd party libraries dependencies.

### 7.4.5. Third party libraries and licensing: technical and legal considerations

The licensing requirements from the PREFORMA project can be summarised as follows:

1. All code (software and libraries) distributed as part of the Conformance Checker, either developed during the project or already developed by the supplier and contributed to PREFORMA, is to be released under GPLv3++ and MPLv2++.

2. All code (software and libraries) distributed as part of the Conformance Checker, either developed during the project or already developed by a third party and contributed by the supplier to PREFORMA, has to be freely available in open source form under generally recognized free software licenses compatible with the GPLv3++ and MPLv2++ to enable redistribution of the whole package under these two licenses.

3. All code (software and libraries) required to compile and/or execute the Conformance Checker in a production environment has to be freely available in open source form under generally recognized free software licenses compatible with the GPLv3++ and MPLv2++ to enable redistribution of the whole package under these two licenses.

DPF Manager is designed as modular software and does include a number of third party software components. These are listed in an annex and their position in the architecture and interaction with the other components are indicated.

Particular highlight has been given to the following technical aspects:

1. The Shell is decoupled (from a logical point of view) from the Conformance Checker
2. There are no third party components in the Implementation Checker, which is the key component of the proposed solution. The source code for this component will be of new creation.
3. The interactions between the Implementation Checker and other components of the platform are by way of exchange of data and not any logical dependency.

The DPF Manager can be seen as a composed work. It will be a work that is the result of the composition or integration of several component parts, without the collaboration of the owners of those parts, in particular the third party FOSS components.

However, from another perspective, Easy Innova will supervise, coordinate and divulge the DPF Manager as a work which is the integration of several parts (the components) of different authors from certain subcontractors whose work is merged into the new “whole”. Thus the DPF Manager will also be a collective work, the exploitation rights in the work as a whole belong to Easy Innova.

Therefore, if there are no obligations imposed by such third party licenses, then the rights holder can freely choose the license of its choice for the solution, and deliver the same to the PREFORMA project under the required dual license (GPL3+/MPL2+).

The complete IPR and licensing report from id law partners can be found in the annexes.

## 8. Benefits of proposed technical design

### 8.1. Interoperability

Our conformance checker has been designed so it can offer four different ways to be integrated with other components and solutions:

1. Command line API: The shell can be used through the command line, in either standalone or client mode (in a client-server architecture). The command-line API can be used by other software that is installed in the same computer, so it can easily be integrated with other conformance checkers, digital image production software such Adobe Photoshop, etc...

2. Socket API: In a client/server architecture, our conformance checker can be started in server mode, accepting connections via sockets and processing requests and responses via well-defined messages.

3. HTTP/REST API: Similar to the socket API, but the communication is over the HTTP protocol. The conformance checker has a well-defined API with HTTP methods, endpoints and expected parameters and data structures.

4. Internal API: The previous three interfaces allow the DPF Manager to interact with any legacy or new system, independent of the programming language and/or architecture. For deeper integrations, our application can also be integrated as a library in any project that uses the same programming language.

By implementing these four different interfaces, we have made sure that our conformance checker can be integrated with pretty much any other system or component, whether they are in the same computer or in a networked environment (private or public network, such as a company LAN or a Software as a Service in the cloud).

### 8.2. Scalability

In terms of software design, by using an Object Oriented language and all its inherent features (inheritance, polymorphism, data abstractions, etc...), we will design & develop the software following the SOLID principles (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion). The SOLID principles were designed to create software that was modular and decoupled, so that it was easy to extend and maintain over time.

In terms of architecture, the conformance checker can be run in either command line mode (local computer), or as a service in a server, listening to a specific port and communicating over HTTP(S). This architecture allows to run as many instances of the conformance checker as one may need (even behind a load balancer), so the process of conformance checking is scalable and can be used in high demand environments.

## 8.3. Portability

Thanks to using Java as the programming language, we can ensure that the solution we will develop will be platform independent and will be able to run on any operating system.

Java programs are never compiled for a specific platform, they are compiled as a Java byte code. This byte code can be deployed without modifications or adaptations to any operating system.

Users will need to have the Java interpreter installed on their computers (they will be able to download it from the project website) to be able to run the conformance checkers. This will be a one-time installation; after it's completed they won't have to install anything else.

We will also use TCP/IP and HTTP as the communication protocols in a client/server scenario, which are supported by all the operating systems.

There will be no part of the project that is platform dependent, so it will be deployable and executable in any environment.

## 8.4. Modularity

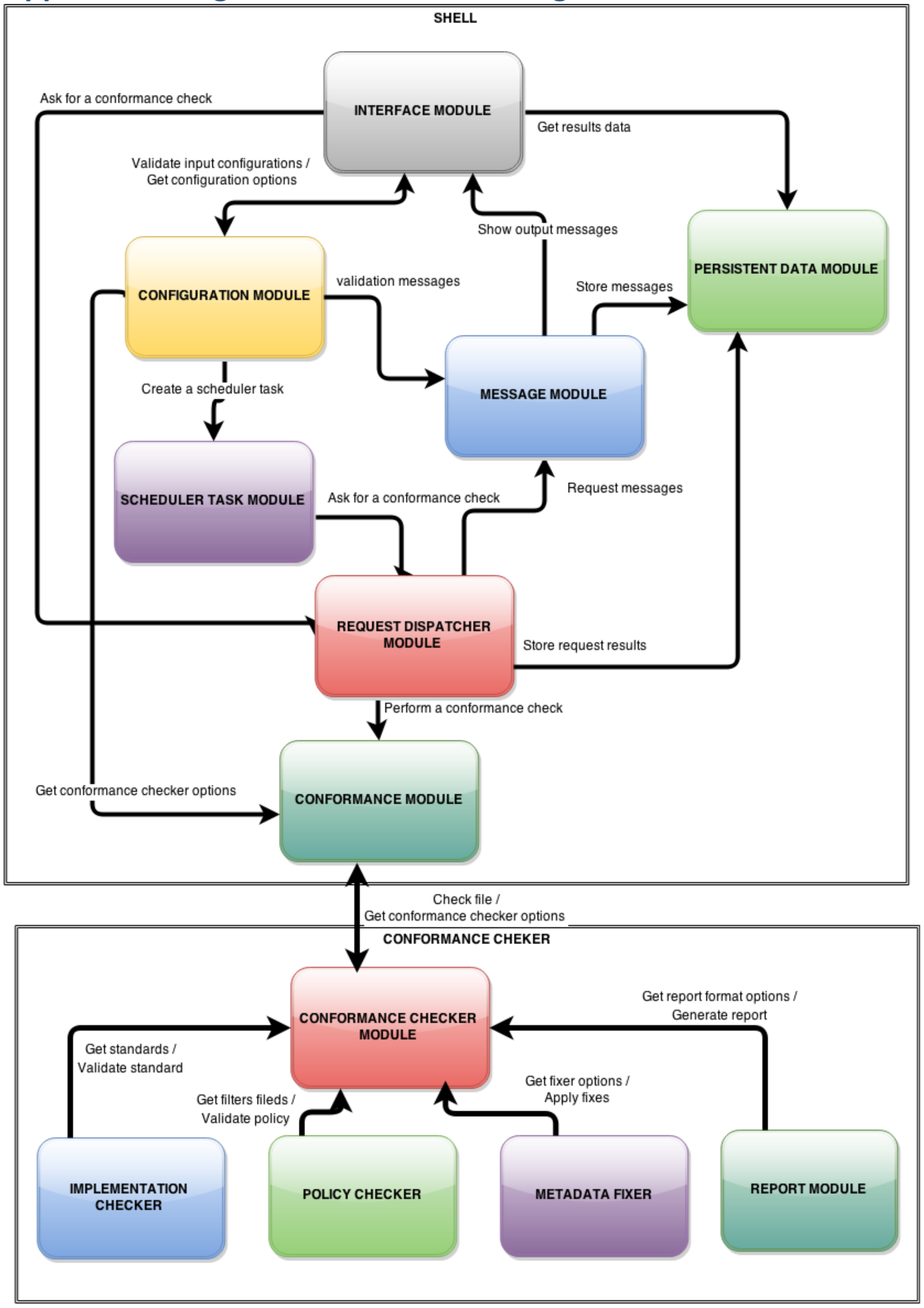
The source code will be structured in a modular way, with well-defined modules and interfaces between them, rather than using modules that are tight-coupled to each other.

This design will make it easier to maintain the different modules of the system, as there will be no hard dependencies.

## 8.5. Deplorability in multiple environments

This architecture allows the software to be run in any configuration possible (stand-alone, client/server, high availability, etc.) . A more thorough description of all the possible use scenarios is described in the Functional Specifications document.

## Appendix A: High level architecture diagram


SHELL
INTERFACE MODULE
Ask for a conformance check
Get results data
Validate input configurations / Get configuration options
Show output messages
PERSISTENT DATA MODULE
CONFIGURATION MODULE
validation messages
Store messages
MESSAGE MODULE
Create a scheduler task
SCHEDULER TASK MODULE
Ask for a conformance check
Request messages
REQUEST DISPATCHER MODULE
Store request results
Perform a conformance check
Get conformance checker options
CONFORMANCE MODULE
Check file / Get conformance checker options
CONFORMANCE CHEKER
CONFORMANCE CHECKER MODULE
Get report format options / Generate report
Get standards / Validate standard
Get fixer options / Apply fixes
Get filters fileds / Validate policy
IMPLEMENTATION CHECKER
POLICY CHECKER
METADATA FIXER
REPORT MODULE

# Appendix B: Third party libraries and licenses

| Domain | Library | Licenses | Module [1] | link |
|---|---|---|---|---|
| **General libs** | Apache commons | Apache 2.0 | General java utilities library (I/O, Math, etc..) | Dynamic linked |
| | Open jdk | GPLv2+CE | Base Java implementation | Dynamic linked |
| **Network library** | Netty | Apache 2.0 | Shell > Interface module > Interfaces > Server interface | Dynamic linked |
| **Json parser** | google-gson | Apache 2.0 | Shell > Configuration module | Dynamic linked |
| **GUI interface** | JAVA Fx | GPLv2+CE | Shell > Interface module > Interfaces > GUI interface | Dynamic linked |
| **PDF** | Apache PDFBox | Apache 2.0 | Shell > Request dispatcher > Global report | Dynamic linked |
| **Documentation** | Javadoc | GPLv2+CE | External tool to generate documentation from source code annotations | |
| **Buider** | Java ant | Apache 2.0 License | External tool to build java apps | |
| **Test** | AntUnit | Apache 2.0 License | External tool to define and execute unit tests. | |
| **Log** | Apache commons | Apache 2.0 | Shell > Message module | Dynamic linked |
| | | | | |
| **Netty dependencies** | base64 | Public Domain License | | |
| | caliper | Apache 2.0 License | | |
| | commons-logging | Apache 2.0 License | | |

| | | | | |
|---|---|---|---|---|
| | deque | Public Domain License | | |
| | jboss-marshalling | GNU 2.1 Licence | | |
| | jsr166y | Public Domain License | | |
| | jzlib | BSD 3 clause License | | |
| | log4j | Apache 2.0 License | | |
| | protobuf | BSD 3 clause License | | |
| | slf4j | MIT License | | |
| | snappy | BSD 3 clause License | | |
| | webbit | BSD 3 clause License | | |