Title: Development of web services for the PABRE system

Course: Master's Degree in Information Technology (MTI)

Student: Fernando Mora Bernabé

Director: Carme Quer Bosor

Department: ESSI

Date: 25/09/2013

Table of content

Table of content 3

Acknowledgements 9

1 Project description 11

1.1 Introduction 11

1.2 Current state of PABRE system 12

1.3 Project goal description 14

1.3.1 Objectives breakdown 16

1.4 Organization of this report 17

2 State of the art of web services 19

2.1 WS-* technologies 19

2.1.1 Simple Object Access Protocol (SOAP) 20

2.1.1.1 Communication model 21

2.1.1.2 Advantages and disadvantages 22

2.1.2 Web Services description 23

2.1.2.1 WDSL 23

2.1.3 Web services register and discovery 24

2.2 RESTful Web Services 26

2.2.1 Architectural principles 28

2.2.2 Advantages and disadvantages 32

2.3 Websockets 33

2.3.1 Advantages and disadvantages 34

3 Study of domain of the PABRE system 37

3.1 Introduction 37

3.2 Classification schema 37

3.3 Software Requirement Pattern structure 38

3.3.1 SRP metadata 38

3.3.2 Requirement forms 38

3.3.3 SRP part 39

3.3.4 Dependencies 41

3.4 PABRE tools analysis 41

3.4.1 Three-tier architecture 41

3.4.2 Used technologies 43

3.4.2.1 Java 43

3.4.2.2 Hibernate 45

3.4.2.3 Apache Derby 47

3.5 Domain class diagrams 47

4 Analysis and design decisions 51

4.1 Requirements analysis 51
4.1.1 Functional requirements 51
4.1.2 Non-functional requirements 52

4.2 Analysis of architectural decisions 52
4.2.1 RESTful web API 52
4.2.2 JSON format 54

4.3 Development platform and technological decisions 57
4.3.1 Java 57
4.3.2 Eclipse 58
4.3.3 Apache Tomcat 58
4.3.4 Jersey 59
4.3.5 Jackson JSON processor 60
4.3.6 Hibernate 62
4.3.7 Apache Maven 63
4.3.8 Git 64
4.3.9 Apiary.io 65
4.3.10 MySQL 66
4.3.11 SQuirreL SQL Client 67
4.3.12 Apache DdlUtils 68

4.4 API design principles 68

4.5 Resources addressing schema 70

4.6 Resources representation design 73
4.6.1 Individual pattern 74
4.6.1.1 Dependency complete representation 77
4.6.1.2 Keyword partial representation 78
4.6.1.3 Requirement Form complete representation 79
4.6.1.4 Fixed Part complete representation 81
4.6.1.5 Extended Part complete representation 82
4.6.1.6 Parameter complete representation 83
4.6.2 Patterns collection 83
4.6.3 Individual pattern version 84
4.6.4 Pattern versions collection 86
4.6.5 Individual Metric 87
4.6.5.1 Integer Metric complete representation 89
4.6.5.2 Float Metric complete representation 89
4.6.5.3 String Metric complete representation 90
4.6.5.4 TimePoint Metric complete representation 90
4.6.5.5 Domain Metric complete representation 91
4.6.5.6 Set Metric complete representation 92
4.6.6 Metrics collection 92
4.6.7 Individual Source 93

4.6.8 Sources collection 94
4.6.9 Individual Schema 94
4.6.10 Schemas collection 96
4.6.11 Individual Classifier 97
4.6.12 Classifiers collection 98
4.7 Resources query parameters 99
5 Development of Pabre-WS 103
5.1 Environment preparation 103
5.2 Apache Maven configuration 104
5.3 Reused code from Pabre-Man 108
5.4 Jersey configuration 111
5.5 Resource classes 114
5.5.1 Used annotations 115
5.5.2 Metrics root resource class 117
5.5.3 Patterns root resource class 118
5.5.4 Sources root resource class 120
5.5.5 Schemas root resource class 121
5.6 DTO classes (Jackson JSON processing) 122
5.6.1 Used annotations 123
5.6.1.1 Declarative Hyperlinking annotations 125
5.6.2 DTO classes implementation 126
5.6.2.1 DTO classes diagram 127
5.6.2.2 GenericObjectDTO 128
5.6.2.3 SourceDTO 128
5.6.2.4 RequirementPatternDTO 129
5.6.2.5 RequirementPatternVersionDTO 131
5.6.2.6 RequirementFormDTO 132
5.6.2.7 PatternItemDTO 134
5.6.2.8 FixedPartDTO 135
5.6.2.9 ExtendedPartDTO 135
5.6.2.10 DependencyDTO 136
5.6.2.11 ParameterDTO 137
5.6.2.12 SchemaDTO 138
5.6.2.13 InternalClassifierDTO 138
5.6.2.14 RootClassifierDTO 139
5.6.2.15 Positionable 140
5.6.2.16 PositionComparator 141
5.6.2.17 MetricDTO 141
5.6.2.18 IntegerMetricDTO 142
5.6.2.19 FloatMetricDTO 143
5.6.2.20 StringMetricDTO 144
5.6.2.21 TimePointMetricDTO 145
5.6.2.22 DomainMetricDTO 147

5.6.2.23 SetMetricDTO 148
5.7 Servlet context listener class 148
5.8 Data layer and Hibernate modifications 150
5.8.1 Hibernate's configuration file mapping file path 150
5.8.2 Hibernate's configuration file load 150
5.8.3 Lazy property 151
5.8.4 MediatorPatterns class methods 152
5.9 Project build and package 154
6 Database migration 155
6.1 MySQL database creation 155
6.2 Database schema and data migration 156
6.2.1 SQuirreL DBCopy plugin 156
6.2.2 Apache DdlUtils 157
6.3 Software tools adaptations to MySQL 159
6.3.1 Hibernate configuration file 159
6.3.2 Mapping file. Identifiers generator 160
6.3.3 Mapping file. Table name references 161
6.3.4 Hibernate JDBC connection pool 162
7 Test client interface for Pabre-WS 167
7.1 pabre.js script 169
7.1.1 jQuery ready function 169
7.1.2 initHTMLElementsState function 170
7.1.3 initHTMLElementsBehaviour function 170
7.1.4 loadSchema function 170
7.1.5 Converting functions 171
7.1.6 loadPattern function 172
8 Project plan and execution 173
8.1 Project planning 173
8.2 Project execution 175
9 Conclusions 179
9.1 Project results 179
9.2 Personal evaluation 180
9.3 Future work 181
10 Bibliography 183
ANNEX I. Pabre-WS API documentation 187

10.1 Schema resources 187
10.1.1 GET /ws/api/schemas 187
10.1.2 GET /ws/api/schemas/{id}{?unbinded,complete} 187
10.2 Classifier resources 190
10.2.1 GET /ws/api/schemas/{id}/classifiers{?complete} 190
10.2.2 GET /ws/api/schemas/{id}/classifiers/{id} 191
10.3 Pattern resources 194
10.3.1 GET /ws/api/patterns?{keyword} 194
10.3.2 GET /ws/api/patterns/{id} 194
10.4 Version resources 196
10.4.1 GET /ws/api/patterns/{patternId}/versions 196
10.4.2 GET /ws/api/patterns/{patternId}/versions/{versionId} 197
10.5 Metric resources 201
10.5.1 GET /ws/api/metrics?{complete} 201
10.5.2 GET /ws/api/metrics?{id} 202
10.6 Source resources 202
10.6.1 GET /ws/api/sources 202
10.6.2 GET /ws/api/sources/{id} 203

Acknowledgements

In this lines I would like to express my gratitude to all the people who has supported me last years to arrive here and make this possible:

I would like to thank all my university partners and friends, partners in crime, that have made my life in the FIB better during these last two years for teaching me a lot of things, for supporting me not throw in the towel working hand in hand with me during the long hard work sessions. Aitor, Emili, Noè, Daria, Michał, Nacho, Marc, Dani, Marcos, Suetonio, Oktay, Bengi, Ismail, Javi and Andrey.

I would also like to thank my flat partners for having been my surrogate family during the two years I have been living in Barcelona and for their continued moral support and encouragement thereafter, making me feel like at home. Aitor, Emili, Laura, Oriol and Carlota.

I would like to express my deep gratitude to Carme Quer for offering me the opportunity of develop this projects and together with Cristina Palomares, for their guidance, their assistance and the continuous support and motivation received from them.

My biggest acknowledgement goes to Ana for understanding me all this time and being supportive and patient despite the distance.

Finally, I will be forever deeply indebted to my parents and my sister María for their support and encouragement throughout my whole academic studies, nothing of what I have achieved would have been possible without them.

1 Project description

1.1 Introduction

The reuse of software requirements may help requirement engineers to elicit, validate and document software requirements and, as a consequence, obtain software requirement specifications of better quality both in contents and syntax.

There are many approaches to reuse in software engineering. Among them, patterns hold a prominent position. "Each pattern describes a problem which occurs over and over again in one environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, 1978). PABRE framework is interested in the use of patterns for the requirements analysis stage, namely Software Requirement Patterns.

The GESSI research group at the UPC is leading the PABRE project jointly with the Public Research Centre Henri Tudor (TUDOR). Their research method consisted on, first of all, building the first version of a requirements patterns catalogue and its underlying metamodel after the study of: the system requirement specifications from several call-for-tender real projects conducted by IT consultants; the background on requirements engineering literature and especially on requirement patterns; some experts' advice. After the construction that catalogue, it was validated in two real projects. During such research, two software tools were developed to support the management and use of patterns.

This research is not a finished work, and one of the ongoing tasks of the group addresses improving the functionality and usability of the tools. The project presented in this document is located within the context of these tasks; its goal is to create a web service that provides users an easy way to access requirement patterns and to create their own tools that make use of such patterns.

1.2 Current state of PABRE system

The current state of the PABRE framework consists in:

- A software requirements pattern catalogue with 66 patterns. The patterns correspond to non-functional requirements (29 of them) and non-technical requirements (37 of them) since these types of requirements are the less sensitive to changes in the problem domain.
- A metamodel that describes the structure of patterns and the catalogue.
- A method for the process of using the catalogue in the requirements engineering stage.

The PABRE system is composed by three subsystems: Pabre-Man, Pabre-Proj and Pabre-Proj-Web. The description of the current PABRE system tools is the following (see Figure 1.1):

- Pabre-Man is a tool that is designed to be used by the requirement engineering expert acting as the catalogue manager to maintain and evolve the software requirement patterns (SRP) catalogue. Its main functionalities are: patterns management, browsing, importation/exportation, printing and catalogue evolution.
- Pabre-Proj is a tool that is designed to be used by requirements analysts during the elicitation of requirements in a software development project, use of external software components, or acquisition of software systems. Its main functionalities are: project management, browsing, importation/exportation, requirements management document generation, or call for tenders document generation and patterns use statistics exportation. Currently, there is also a web version of this tool, in a way that requirements analysts can access and edit software requirements projects from everywhere

- PABRE-Proj-Web is a web version of PABRE-Proj that offers the same functionalities as the desktop version, and that has as a goal to facilitate the access and edition of software requirement projects from everywhere without needing one specific tool.

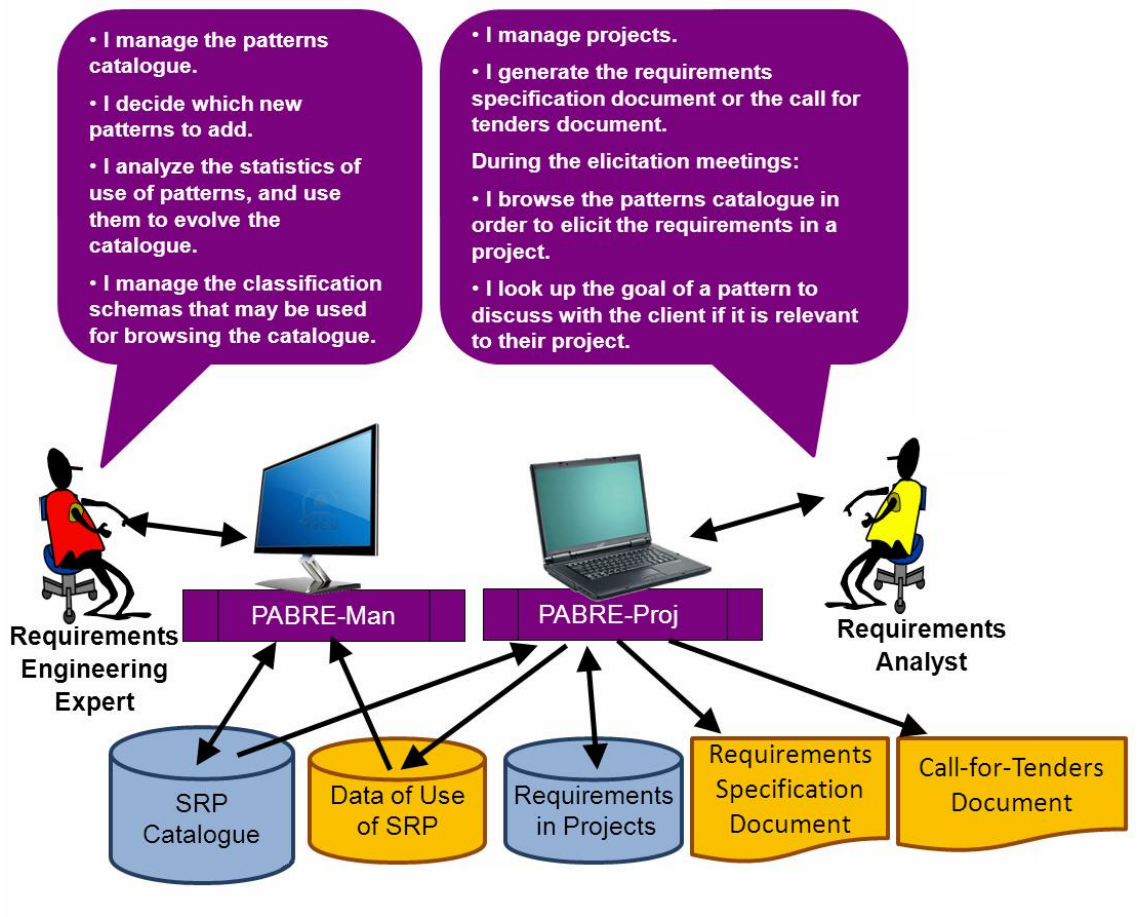


Figure 1.1. The PABRE system

In Figure 1.2 it can be seen a screenshot of the PABRE-Man tool. At the left there is the patterns window. At the right there is the project window. On both windows the same classification schema for classifying SRP and the requirements is used. The idea is that requirements in a project may come from the application of an SRP, or may be related or associated to an SRP, or may be new that means do not related with any SRP.

The SRP catalogue may evolve by analysing the projects information and statistics of use of SRP. For instance: SRP that are never used may be considered to be removed from the catalogue; requirements defined as associated to SRP can be promoted as new parts of SRP; and new requirements, do not related with any SRP, can be considered to be converted to new SRP.

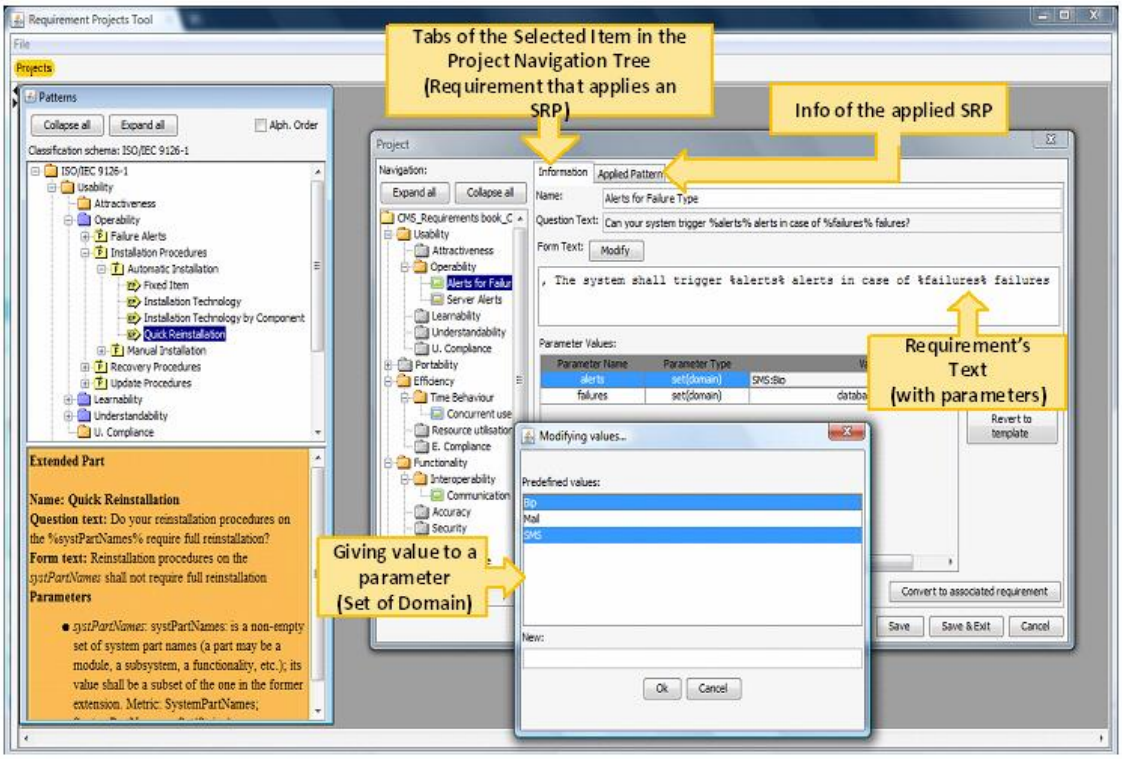


Figure 1.2. Screenshot of PABRE-Man tool

1.3 Project goal description

The main goal of this project consists on developing a web service (PABRE-WS) to allow different Requirement Management Tools (RMT) to access SRP catalogue information enabling to create or to adapt client RTM applications that makes use of the SRP catalogue such as PABRE-Man or PABRE-Proj (see Figure 1.3).

Specifically, these RMT should be able to access the SRP catalogue, the different classification schemas defined for the catalogue and also to allow to search for SRPs over the catalogue. In order to test the optimal behaviour of the Pabre-WS, a client side interface will be developed.

The project includes the creation of a semi automatized method to perform data migration of the current tools of the system to a new centralized DBMS (MySQL) from the current DBMS (Apache Derby) in order to allow the system to work alternatively on a centralized database with a better support in the servers that currently host the PABRE system providing higher availability, reliability and performance.

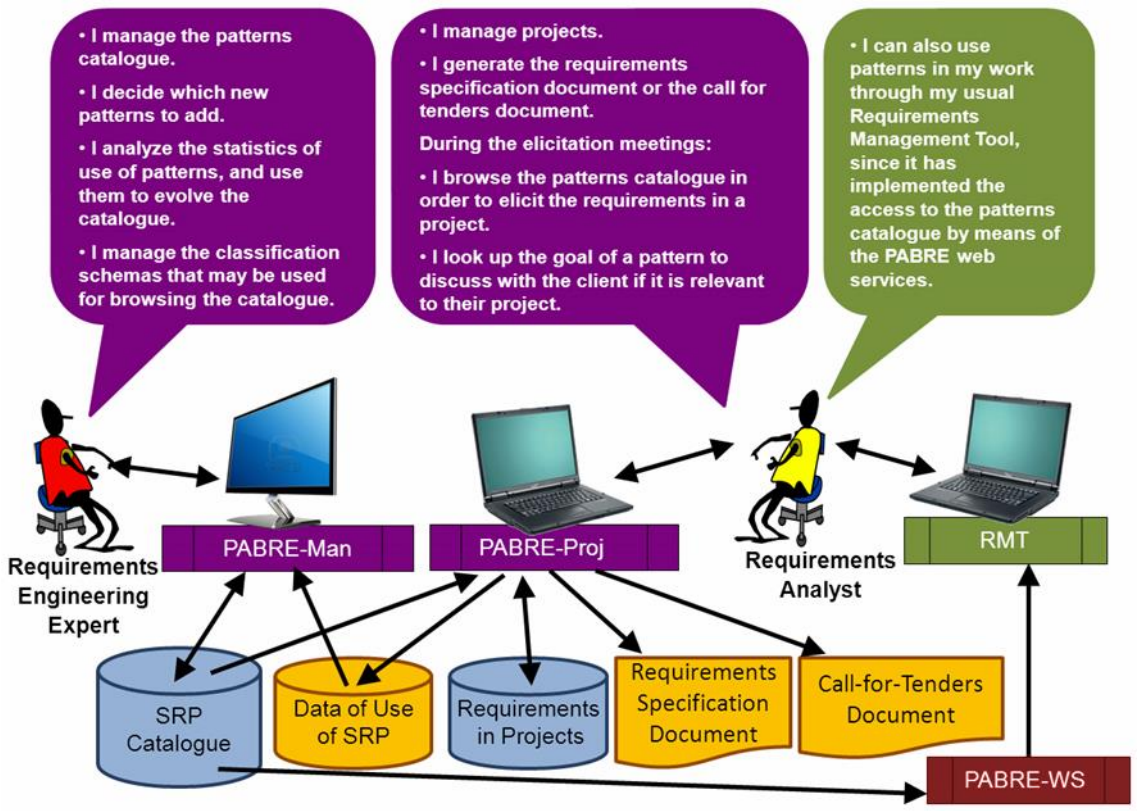


Figure 1.3. Pabre-WS in the mark of PABRE system

The development and inclusion of Pabre-WS in the framework has two motivations:

- Allow different clients acting as requirements analysts to access the SRP catalogue knowledge not needing a specific software tool such as PABRE-Proj or PABRE-Proj-Web, so that they can create or adapt their own tools to manage this knowledge and be free to use it in their own way.
- Create a new layer (see the new architecture in Figure 1.4) between the client tools and the SRP database, hiding this database and avoiding direct access to it from client tools, limiting the range of operations that a client can perform over the catalogue increasing the security of the framework.

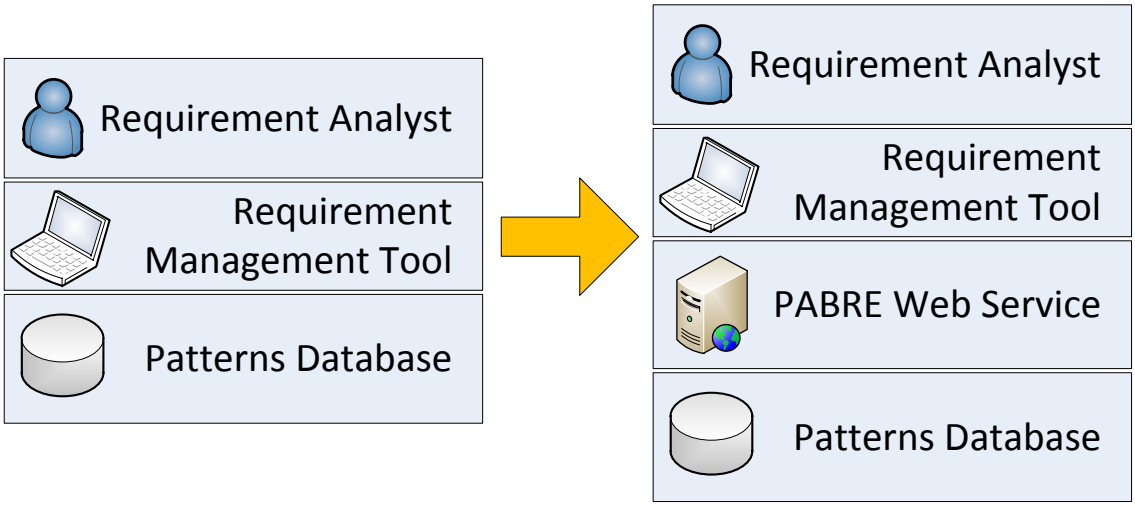


Figure 1.4. PABRE architectural evolution

This web service will allow only query operations over the catalogue, since it is considered that clients of the catalogue are not responsible of managing and maintain the catalogue and the requirement engineering expert is a trusted user of the system that will continue using Pabre-Man tool to manage the catalogue from the private network of the framework so no extra security is needed.

1.3.1 Objectives breakdown

To accomplish this goal, it has been divided in the following set of required objectives:

1. Perform the state of the art study, analysis and comparison of current web services technologies and architectures.
2. Analyse the data and domain model of the current tools Pabre-Man, Pabre-Proj and Pabre-Proj-Web, to decide the reusability level of their code in Pabre-WS.
3. Take the architectural decisions and select the framework used for the web service technology.
4. Analyse and select the available environment tools and libraries suitable for the implementation phase and learn to configure and use them.
5. Design the Pabre-WS web service.
6. Implement the Pabre-WS web service.
7. Implement of the test client-side interface.
8. Automatize database migration from Apache Derby (current DBMS) to MySQL and vice versa and adapt current software (Pabre-Man, Pabre-Proj and Pabre-Proj-Web) and the new Pabre-WS web service to the new DBMS.

9. Elaborate of API documentation.
10. Write final report of the project.

1.4 Organization of this report

The structure of the project report is the following (see the relationships between goals above and sections in Figure 1.5):

In section 2 the result of the state of the art of web services technology is presented and each technology is briefly analysed with its advantages and disadvantages.

In section 3 the study of the state of PABRE system before beginning the project is presented. The relevant information about how SRP are managed and stored relevant for the project and the conceptual schema is described. Also the technologies that the system uses and that have to be considered in this project are described.

In section 4 it is presented the requirements of the Pabre-WS web service. The relevant architectural decisions taken for the design and their motivations are described. The list of the tools and libraries used in the scope of this project and their description and motivation is explained and the design principles and the full description of the design process is documented.

In section 5 the development process of Pabre-WS has been documented, describing the code reused from previous PABRE tools, the configuration of the environment and the implementation tasks performed to develop the Pabre-WS web service showing the resulting classes derived from the design described in section 4.

In section 6 the DBMS migration tasks is described, explaining all the necessary steps, the alternatives, the problems found during the process, the description and configuration of the required tools and the modifications made over the software tools to adapt both Pabre-WS and previous PABRE tools to the new DBMS (MySQL).

In section 7 it is described the implementation of the test client-side interface, its features and resources used.

In section 8 the project plan and execution is described showing the previous planning at the beginning of the project and the final execution schedule followed assigning the temporal cost of each task.

In section 9 the conclusions of the project elaboration are presented, analysing the results and success of the project, my personal evaluation and my suggestions about the future work that could provide interesting improvements to PABRE system.

In annex 1 it is shown the elaborated API documentation that act as a guide for web service clients to develop or adapt their applications to access the SRP catalogue.

	Sect.2	Sect.3	Sect.4	Sect.5	Sect.6	Sect.7	Sect.8	Sect.9	Sect.10	ANNX.I
Object.1	X									
Object.2		X								
Object.3			X							
Object.4			X							
Object.5			X							
Object.6				X	X					
Object.7						X				
Object.8					X					
Object.9										X
Object.10	X	X	X	X	X	X	X	X	X	X

Figure 1.5. Project objectives by report sections

2 State of the art of web services

The aim of this section is to provide an overview and an analysis of standards in the field of so called WS-* web services as well as to discuss the different benefits they offer. In addition, it will look at the alternatives architectures and technologies for developing and deploying web services such as RESTful web services and Web Sockets.

2.1 WS-* technologies

Several Web services specifications have been developed or are still being developed in order to enlarge Web Services functionalities. These specifications are generally referred to as WS-*.

WS-* services are generally based on the following layers:

- Service transport: The layer that delivers messages between applications. This layer usually implements hypertext transfer protocol (HTTP), but can be any other protocol such as FTP, SMTP, etc.
- Service messaging: The layer responsible for encoding messages in a common XML format. Currently, this layer includes SOAP (Simple object access protocol) that is a simple XML-based messaging protocol responsible for transferring data between different web services.
- Service description WSDL: The purpose of this layer is to define the public interface of a specific web service. Currently, service description is elaborated through the Web Service Description Language (WSDL) which is based on XML.
- Service discovery: The service discovery layer registers services into a common repository and provides an easy publish/find mechanism. This layer is often implemented via Universal Description, Discovery, and Integration (UDDI).
- Service orchestration: The topmost service layer is in charge of the execution logic of web services based applications by determining their control flows.

The typical architecture of a WS-* web service is a software built using a specific programming language that is published using a WSDL interface that can be invoked by a consumer “client” using this interface. These web services are presented to clients as

a set of operations that provide business logic on behalf of the provider. On the client side, a remote object that represents the remote service must be generated allowing clients to invoke the operations defined on the server side. These operations are sent as messages using the SOAP protocol, but developers do not care about creating or parsing SOAP messages, that task is performed by the web service's APIs runtime system. Finally web services are published registering them in a service directory using UDDI.

In the following sections all these concepts are described considering their advantages, disadvantages and consequences of using them.

2.1.1 Simple Object Access Protocol (SOAP)

The conventional distributed communication link, in Traditional distributed object communication protocols, is was typically implemented under a distributed object model and needed the deployment of libraries. To solve these problems, Simple Object Access Protocol (SOAP) was created. It allows interoperability between a wide range of programs and platforms. In this way, applications can be accessed by a wide range of users (Papazoglou, 2008).

Nowadays, the messaging protocol currently used by WS-* services is SOAP. SOAP enables separate distributed computing platforms to interoperate. This aim is accomplished by following the same principles as other successful web protocols: simplicity, flexibility, firewall friendliness, platform neutrality as well as XML based messaging.

SOAP messages can be built on different protocols, although it is usually exchanged through HTTP (see Figure 2.1) which is the protocol used by Web browsers to access Web resources. Other protocols that may be used are SMTP or FTP (Papazoglou, 2008).

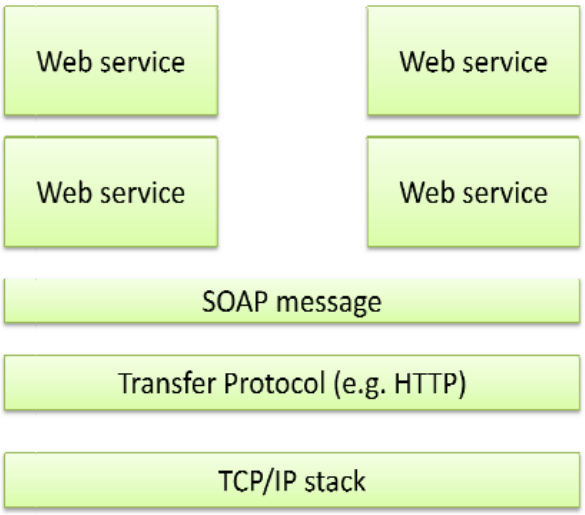


Figure 2.1. SOAP architecture stack.

HTTP constitutes and receives the SOAP messages (Papazoglou, 2008). SOAP's role is to define how a message is formatted but not how the message is delivered.

2.1.1.1 Communication model

The SOAP communication model is defined by its communication style and its encoding style. SOAP supports two possible communication styles:

- RPC and document (message).

  RPC-style web services are used as remote objects on the client application side. Clients send their request as a method call. The method returns a response message (Papazoglou, 2008). This information is formatted as sets of XML elements loaded into a SOAP message.

- Document (message)-style Web services

  SOAP supports documents exchange for any kind of XML data (see Figure 2.2). The client sends the whole document to the provider instead of sending a set of arguments (Papazoglou, 2008).

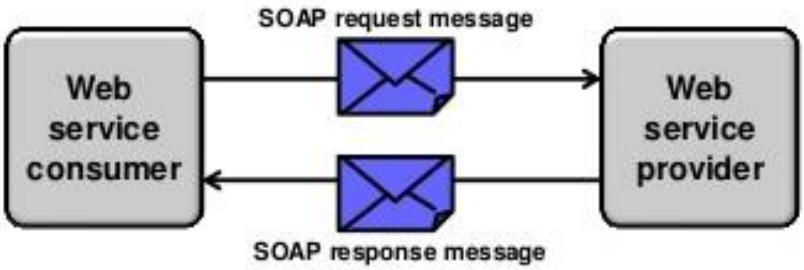


Figure 2.2. SOAP message interaction.

2.1.1.2 Advantages and disadvantages

There are several advantages of using SOAP (Papazoglou, 2008):

- Simplicity: SOAP is simple as it uses XML that is well structured and easy to parse.
- Portability: SOAP is platform-independent and thus portable.
- Firewall-friendliness: SOAP is capable of getting past firewalls which are totally blocking for other protocols. This is possible due to using the HTTP protocol.
- Use of open standard: SOAP is based on the open Standard XML to format data. As a consequence, SOAP becomes easily extendable and well supported.
- Interoperability: SOAP relies on open instead of vendor-specific technologies and thus enables distributed interoperability and loosely coupled applications.
- Resilience to changes: It is unlikely that future modifications of SOAP infrastructure will have any impact on applications using the method, as long as no significant serialization changes are made to the SOAP specification.

There are, however, several aspects of SOAP which can be viewed as disadvantageous (Papazoglou, 2008):

- SOAP is stateless which implies that the requesting application has to reintroduce itself to other applications if additional connections are needed as if it was connected for the first time.
- SOAP serializes by value and does not serialize by reference.
- SOAP was at the beginning mainly based on HTTP. This imposed a request/response architecture which did not suit all situations. As HTTP is relatively slow, the performance of SOAP was affected.

2.1.2 Web Services description

The process of publishing a Web service involves creating a software system and making it accessible to potential consumers. Web Service Providers need to offer a Web service URI together with an interface description using the Web Services Description Language (WSDL) so that the consumer will be able to use the service.

2.1.2.1 WDSL

"WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate" (W3C, s.f.).

A WSDL document contains a formal definition of the service interface so that requestors who intend to invoke the service provider know how to build the messages. Additionally, it provides the physical location of the service.

A WSDL service description can be separated into two parts (see Figure 2.3) (Erl, 2006):

1. Abstract description: Provides information about the interface characteristics of the Web service without any description about the technology used to implement the web services or used to send messages. So, the integrity of the web service is preserved despite changes on the underlying technology. It contains several information like types, message and port type.
2. Concrete description: The concrete description part describes the connection to the real implementation of the web services where is its logic. This description contains three parts: binding, port and service.

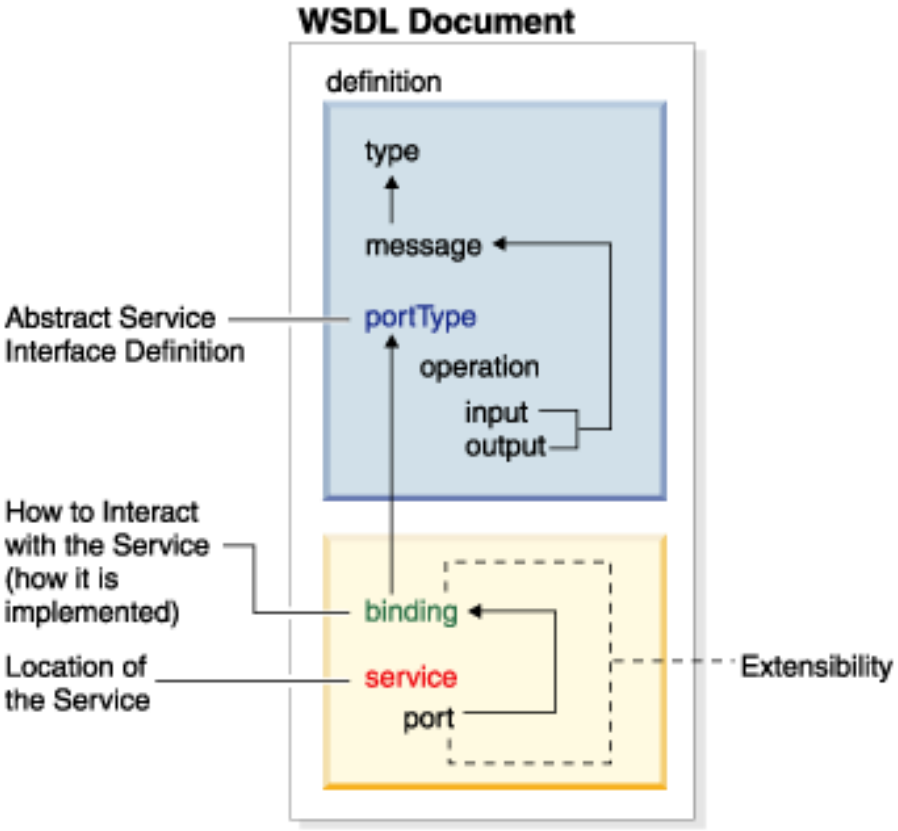


Figure 2.3.WSDL Document Schema

2.1.3 Web services register and discovery

Web services provide access to systems over the Internet using standard protocols. In the most typical scenario there is a Web Service Provider that publishes a service and a Web Service Consumer that uses this service. Web Service Discovery is the process of finding the correct Web Service for a given task (see Figure 2.4).

To make this process easier, optionally, a provider can register a service with a Web Services Registry like the Universal Description Discovery and Integration (UDDI) or publish additional documents in order to allow the consumer to discover them like the Web Services Inspection Language (WSIL) documents. In this way, consumers can search Web Services manually or automatically. The UDDI or WSIL technologies provide simple search APIs or a web GUI to help to find Web services.

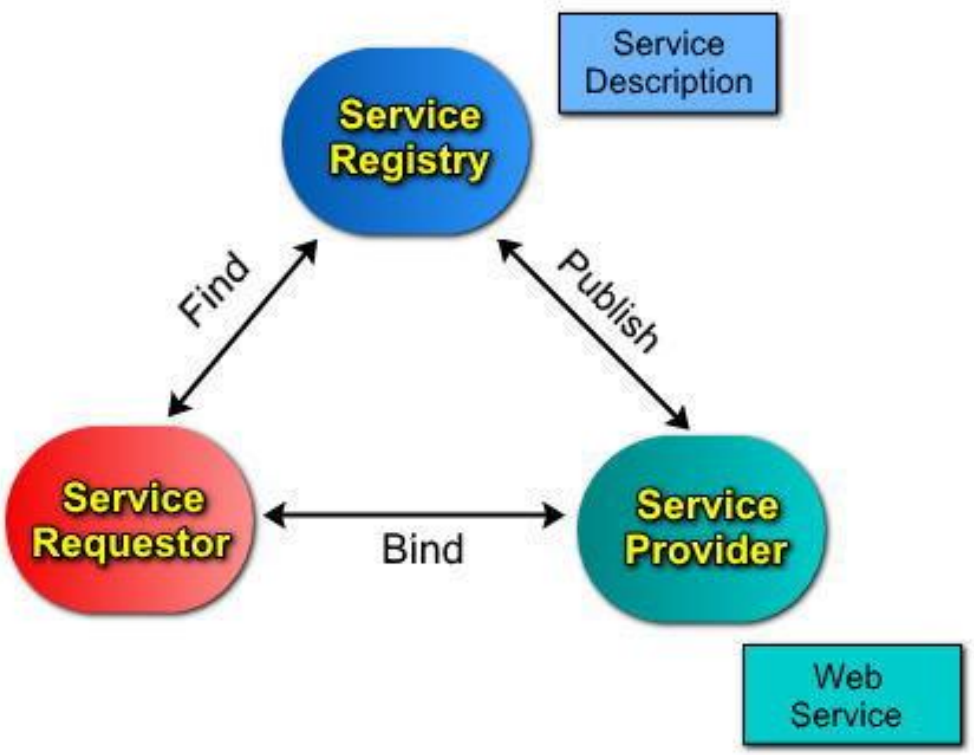


Figure 2.4. Web services register and discovery interaction model

Directory services are an important component of Web Services because they allow businesses to collaborate dynamically over the Web using open protocols and XML.

Directory services provide information about the information point for components and participants of web services and enables them to locate each other. Directories organize and provide web services' location and description details and send them to any requestor.

Two different kinds of directories are available: name servers referred as "white pages" in which entries are defined and found by their name as well as so called "yellow pages", where entries are defined and found by their characteristics and functions (Singh, Munindar & Huhns, Michael, 2005).

Nowadays, two main standards for directories have already been defined: ebXML (Electronic Business using XML) registries and UDDI (Universal Description, Discovery and Integration) registries.

The searches can be done only by keywords, like a service's name, provider, location or business category. In contrast with UDDI registries, ebXML registries permit at least SQL-based queries on keywords (Singh, Munindar & Huhns, Michael, 2005).

A summary of the comparison among the two registries can be found in Figure 2.5.

UDDI Registry	ebXML Registry and Repository
Contains no repository. Unable to store content. Capable only of storing metadata about (or pointers to) content.	Has an integrated Registry and Repository. Able to store content as well as metadata.
Design center lists businesses and services, similar to yellow/white pages.	Design center provides secure, federated information management of any type of artifact.
Protocols and information model is focused and specific.	Protocols and information model is generic and extensible.
Supports multi-registry topologies using replication of every transaction to all participating registries.	Supports multi-registry topologies using loosely coupled federations with optional selective replication.

Figure 2.5. UDDI and ebXML registry comparison summary

2.2 RESTful Web Services

REST the abbreviation of "Representational State Transfer" designates a software architecture style used for distributed network applications. REST uses a stateless, client-server, cacheable communications protocol which is almost always the HTTP protocol. Its original feature is to work by using HTTP in order to make more simple calls between machines than other existing complex mechanisms such as CORBA, RPC or SOAP. (Elksteing, 2008)

REST-style architectures are composed by clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses (see Figure 2.6). Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

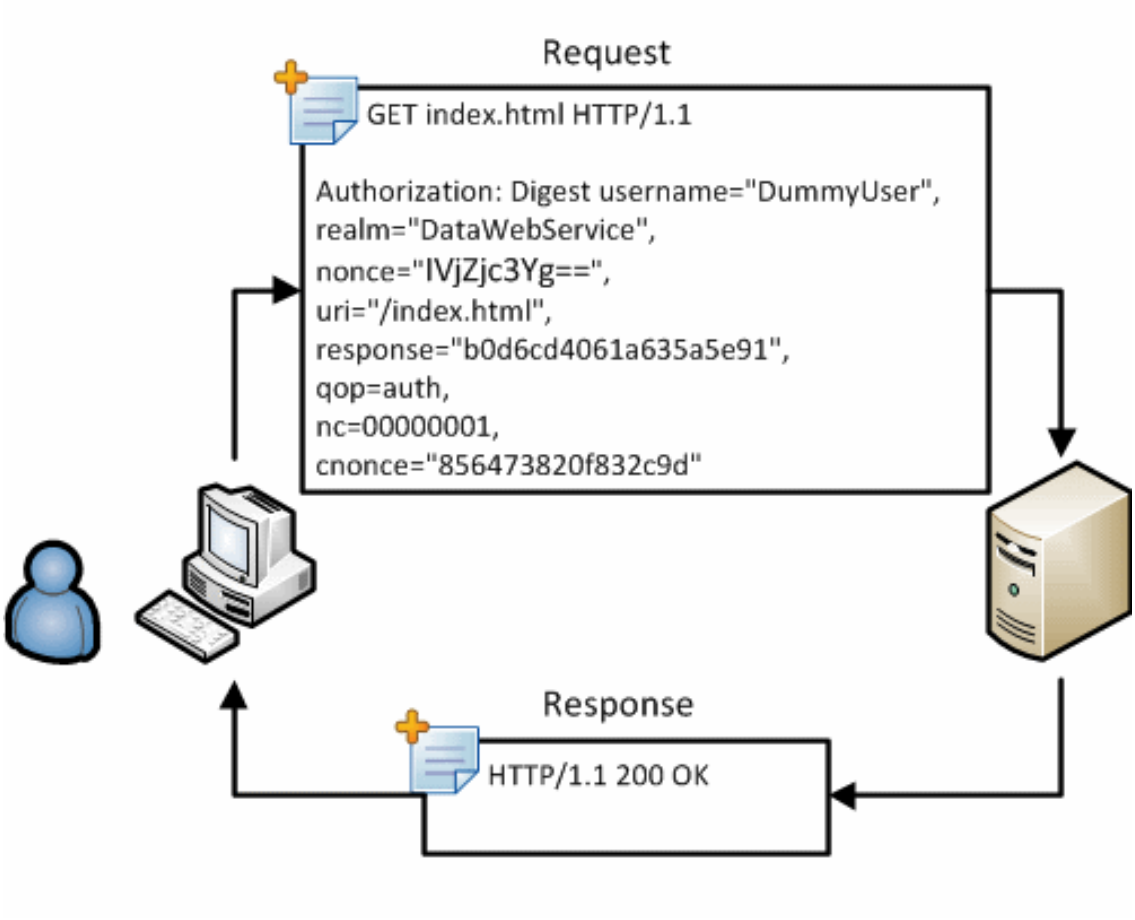


Figure 2.6. RESTful request-response model

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. In this way, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations (Elksteing, 2008).

Unlike SOAP-based web services, there is no "official" standard for RESTful web API and there will never be a W3C recommendation for REST. This is because REST is an architectural style, unlike SOAP, which is a protocol.

Even though REST is not a standard, a RESTful implementation such as the Web uses standards like HTTP, URI, XML, etc. and while there are REST programming frameworks, working with REST is so simple that you can often manage your own implementation with standard library features in several languages (see Figure 2.7).

REST does not offer security features, encryption, session management, QoS guarantees, etc. But these can be added by building on top of HTTP, for example,

username/password tokens are often used and for encryption and REST can be used on top of HTTPS (secure sockets).

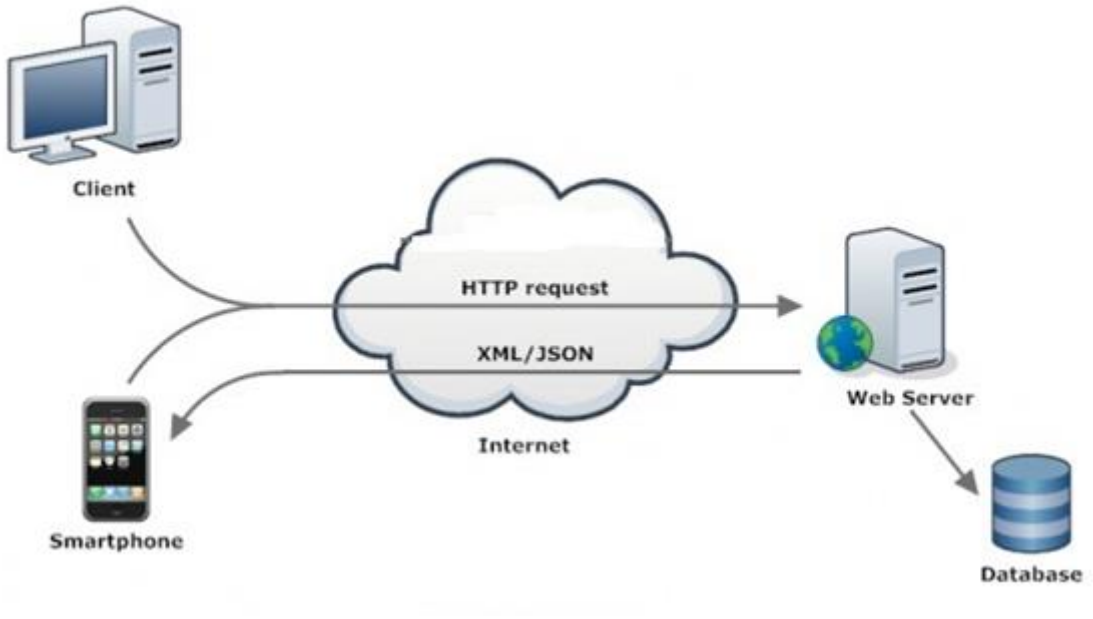


Figure 2.7. Typical REST web server architecture

2.2.1 Architectural principles

The REST architectural style describes the following six constraints applied to the architecture:

1. Client-server.

   Clients are separated from servers by a uniform interface. This means that clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved and servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable.

   Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

2. Stateless communication.

   Client context must not being stored on the server between requests. Each request from any client contains all of the information necessary to service the request.

Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, e.g., URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

3. Cacheable.

As HTTP transactions on the World Wide Web, clients and proxies can partially or completely cache responses to eliminate some client-server interactions, improving scalability and performance.

Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests.

4. Layered system.

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. They may also enforce security policies.

Any number of connectors (e.g., clients, servers, caches, tunnels, etc.) can mediate the request, but each does so without being concern about anything but its own request (see Figure 2.8).

An application can interact with a resource by knowing two things: the identifier of the resource and the action required. It does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between it and resource (Mulloy, 2013).

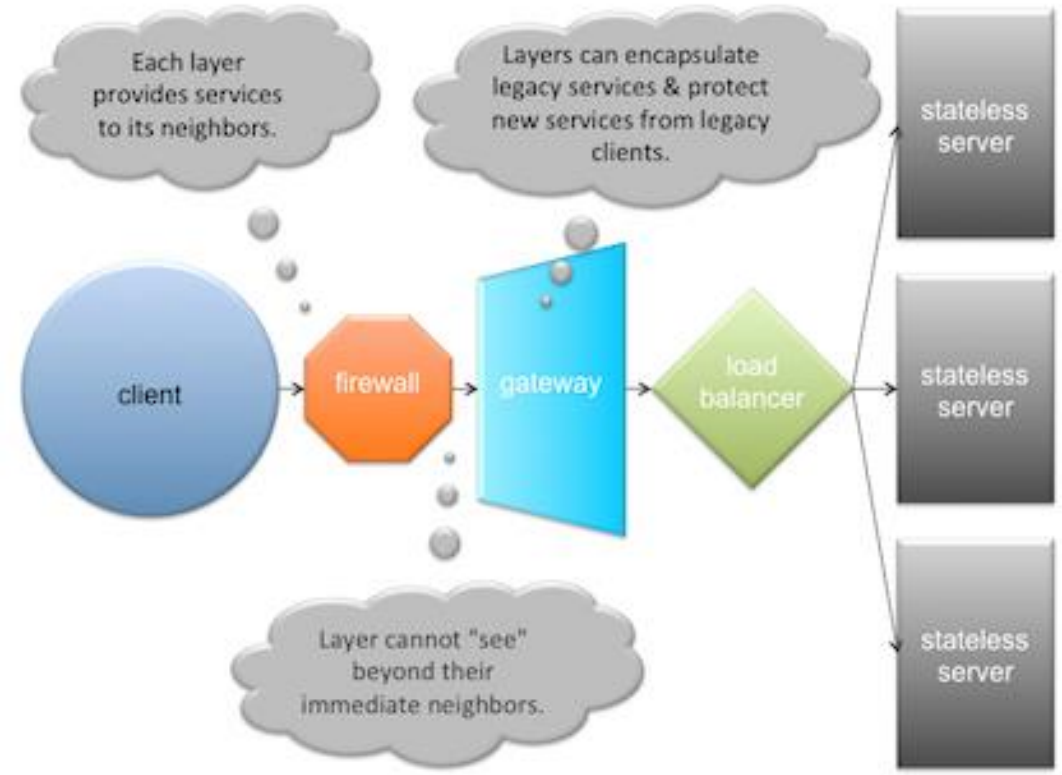


Figure 2.8. Layered System Architecture Schema (Mulloy, 2013)

5. Code-on-demand (optional).

Servers are able to temporarily extend or customize the functionality of a clients by transferring them some logic that they can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

6. Uniform interface.

There is a uniform interface between clients and servers that simplifies and decouples the architecture and enables each part to evolve independently. This uniform interface is guided by this four principles (Pautasso, Zimmermann, & Leymann, 2008):

- Identification of resources.

  Every Individual resource must be identified by one (uniform) resource identifier mechanism in requests, in this case resources are identified though HTTP URIs.

The resources themselves are conceptually separate from the representations that are returned to the client. The server could, for example return a HTML, plain-text, PDF, JPEG, XML or JSON that represents one and the same specific resource expressed in different languages or character encodings depending on the details of the request and the server implementation. All of these representations for a given resource must be referenced under the same URI.

- Manipulation of resources through these representations.

When a client has a representation of a resource, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

There is a common universal semantics for the actions performed over the resources. They mean the same for all resources and resources are manipulated through the use of an action and the exchange of representations of the resources.

The requests actions are based in the set of HTTP request methods. The most notably are GET, POST, PUT and DELETE. PUT creates a new resource, which can be then deleted using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.

- Self-descriptive messages.

Resources are decoupled from their representation so that their content can be accessed in a variety of formats (e.g., HTML, XML, plain text, PDF, JPEG, etc.). Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability.

- Hypermedia as the engine of application state (HATEOAS)

Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for a simple predefined fixed set of entry points to the application, the web service must not assume that any client knows how to access particular resources beyond those described in representations previously received from the server (see Figure 2.9).

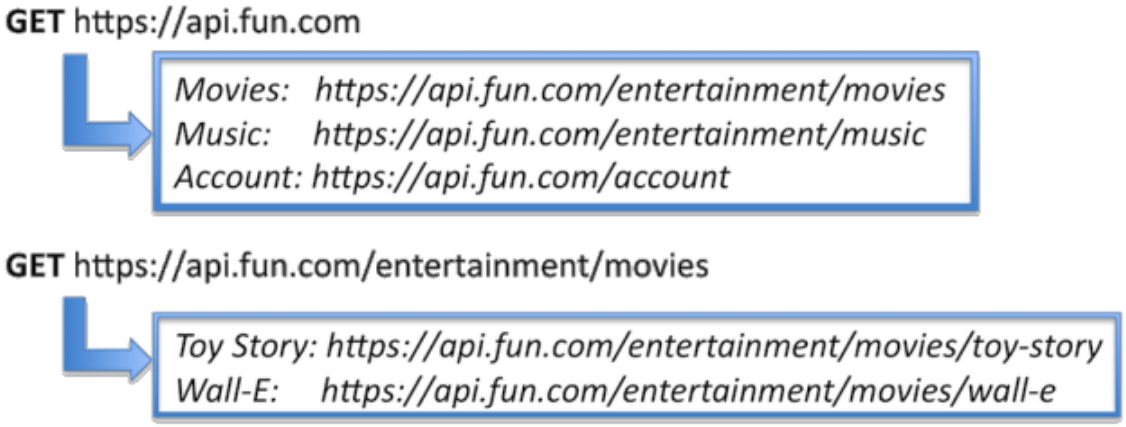


Figure 2.9. Hypermedia as the engine of application state example

2.2.2 Advantages and disadvantages

RESTful has some aspects which can be viewed as positive, including the following (Pautasso, Zimmermann, & Leymann, 2008):

7. RESTful Web services appear to be simple because REST applies many existing well-known standards (HTTP, XML, URI, and MIME) and need only infrastructure that has already become ordinary.
8. HTTP clients and servers are compatible with all programming languages and operating system/hardware platforms, and the default HTTP port 80 is usually left open by default in most firewall configurations.
9. Only a small effort is needed to build a client of a Restful service. Services can be tested using simply a mere web browser and the development of client software becomes superfluous.
10. REST allows discovering Web resources without any discovery or registry repository.

RESTful services has also some negative aspects, including the following (Pautasso, Zimmermann, & Leymann, 2008):

- Encoding a large amount of input data in the resource URI is impossible because the server either refuses such requests or crashes
- It may also be challenging to encode complex data structures into URI as there is no commonly accepted marshalling mechanism. Inherently, the POST method does not suffer from such limitations.
- Unlike SOAP-based web services, which have a standard vocabulary to describe the web service interface through WSDL, Restful web services currently have no a standard grammar. An agreement has to be established between the service consumer and service producer. RESTful services can be described using Web Application Description Language (WADL). It is an XML-based file format that provides a machine readable description of REST web services. It is the REST equivalent of SOAP's Web Services Description Language (WSDL), but W3C consortium has no current plans to standardize it (World Wide Web Consortium., 2009) and it is not yet widely supported.

2.3 Websockets

WebSocket is a web technology providing full-duplex communications channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as in 2011, and the WebSocket API is not still standardized by the W3C (World Wide Web Consortium., 2009).

WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.

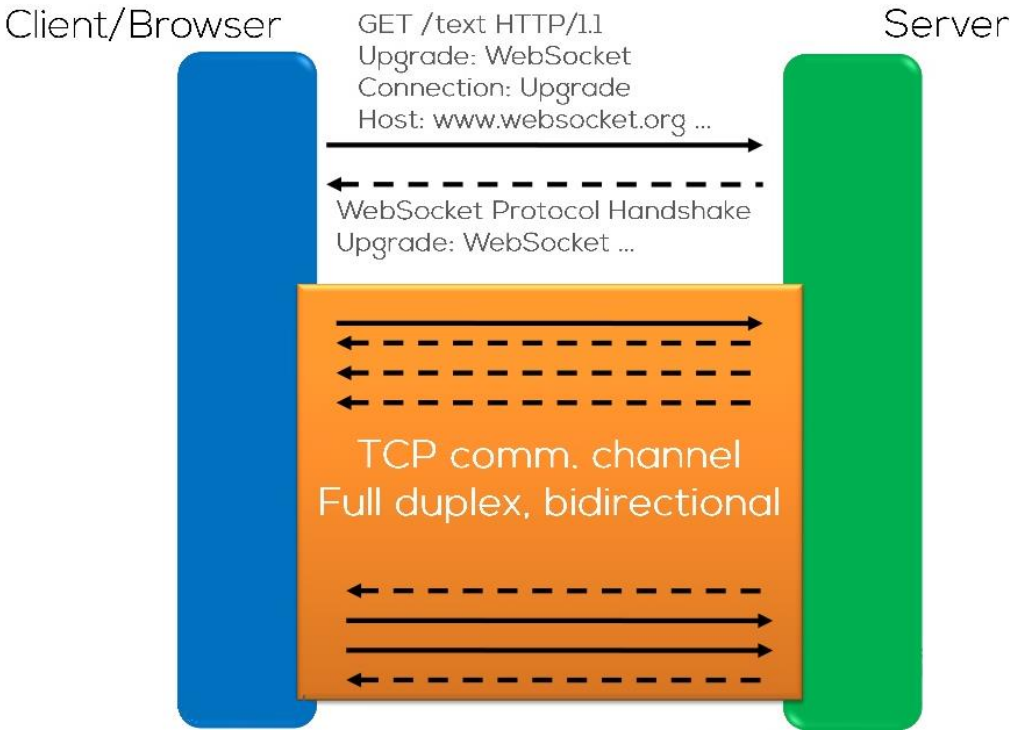


Figure 2.10. Websockets connection

The WebSocket protocol makes possible more interaction between a browser and a web site, facilitating live content and the creation of real-time games (see Figure 2.10). This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way a two-way (bi-directional) ongoing conversation can take place between a browser and the server.

In addition, the communications are done over TCP port number 80, which is suitable at environments where non-standard Internet connections are blocked using a firewall. WebSocket protocol is currently supported in several browsers including Google Chrome, Internet Explorer, Firefox, Safari and Opera. WebSocket also requires web applications on the server to support it.

2.3.1 Advantages and disadvantages

Websocket protocol has some advantages over pure HTTP protocol (this includes REST web services):

- HTTP is a synchronous communication protocol based entirely on "request/response". That is, there is no easy way to open continuous

communication between a client and a server. You have to send a request and wait for the response. There have been a number of techniques developed to make HTTP more "full-duplex", these include long-polling, Comet, HTTP streaming and recently, web sockets. While REST over HTTP has transformed "over-the-internet" communication, it is usually a poor choice for high-throughput or asynchronous communication. For example, most software programs do not communicate to their databases over HTTP. It's typically too slow because of HTTP overhead.

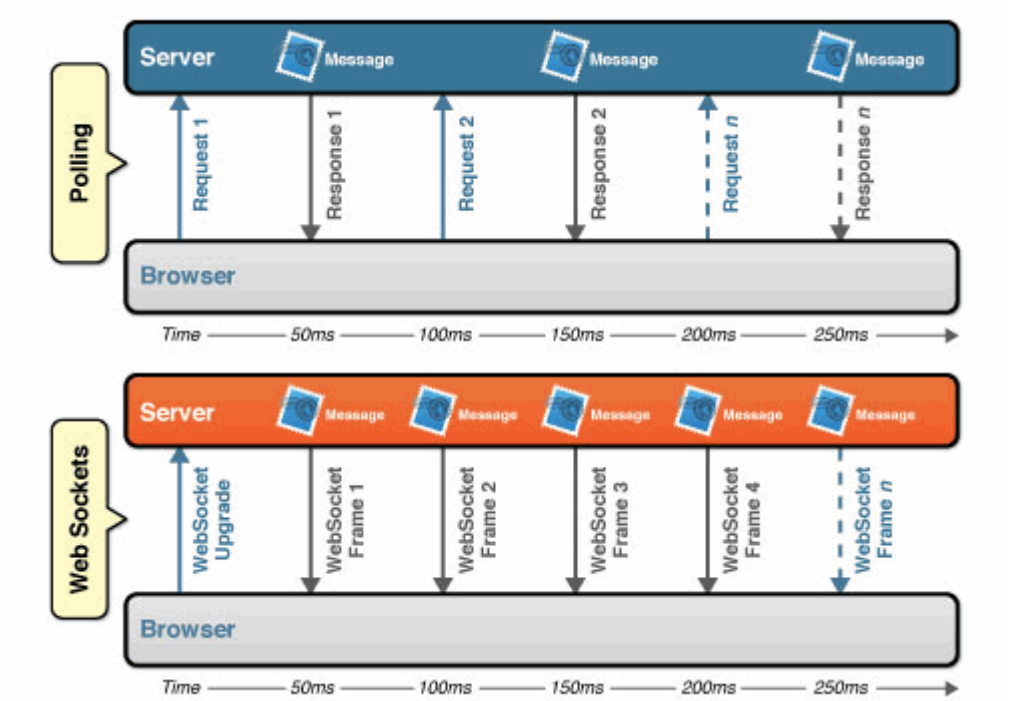


Figure 2.11. Latency comparison between the polling and WebSocket applications. (Lubbers, Greco, & Corporation, 2013)

- By being full duplex–meaning, a program can both ask a question and listen to a response simultaneously–and truly asynchronous, they can not only speed up software by "doing more things at once", they can provide a more efficient pipe to send information through. In Figure 2.11 the latency comparison between the polling and the WebSocket application is presented. WebSocket connections are intended to be more persistent than HTTP connections. If you only want to receive an update every 30 minutes, you will want to go with HTTP. If you want to receive updates every second, a WebSocket might be a better option, because establishing an HTTP connection takes a lot of time.

On the other hand, some considerations should be considered if the decision of use Websocket is taken:

- WebSockets are simply a communication protocol, they do not define a structure by itself. All the benefits that came with REST (structure, human readability, self-description, uniform interface) is abandoned for the sake of efficiency.
- Websockets essentially allows you build your own proprietary protocols that may or not be better than a standard one, with all the typical advantages and disadvantages of this fact: possibly better performance, possibly better suited to the specific task at hand, less standardized, not widely implemented, etc.
- In the long run, HTTP (used in a way aligned with its architectural goals) have more benefits for loosely coupling systems, using Websockets will imply a better efficiency sacrificing loosely coupling.

3 Study of domain of the PABRE system

3.1 Introduction

Software Requirements Patterns (SRP) try to help solving some problems that usually in requirements engineering. Specifically: the definition of ambiguous, incomplete, incoherent requirements; and the definition of requirements without any systematic expression guide. By using these SRP, it is possible to have a standard framework to create and establish requirements helping to create well-expressed requirements improving their quality and saving time, cost and effort in the elicitation phase.

In this section a study and analysis of the PABRE system is presented. Specifically the focus will be in the part of the system relevant for the project here reported.

3.2 Classification schema

In order to allow an easier understanding and reusing of SRP during elicitation processes, every SRP has to be indexed in some hierarchical classification schema of the SRP catalogue. In PABRE the same SRP can be classified in more than one schema (see Figure 3.1 where three classification schemas are presented for the same catalogue) in order to allow the use of one specific pattern in different classification schemas and allowing requirements engineers to customize their own perspective of the available SRP.

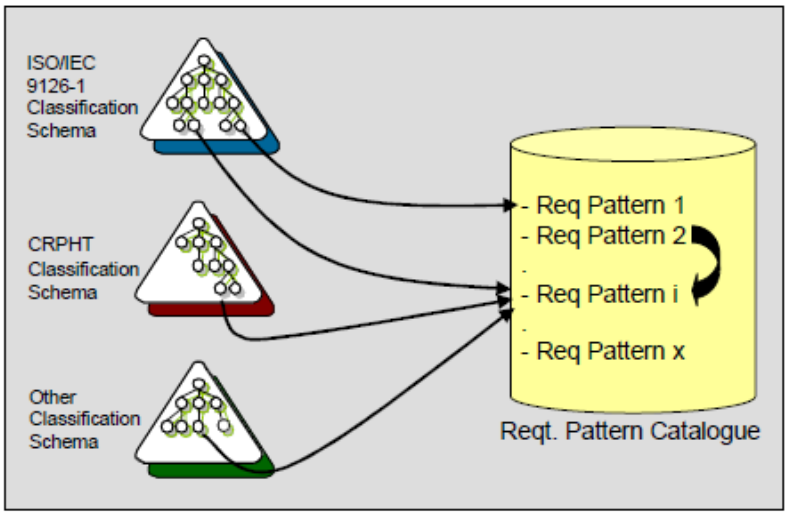


Figure 3.1. Requirement Pattern Catalogue organization

3.3 Software Requirement Pattern structure

The SRP structure is described in Figure 3.2 as a UML conceptual model, and one example that follows such structure can be seen in Figure 3.3.

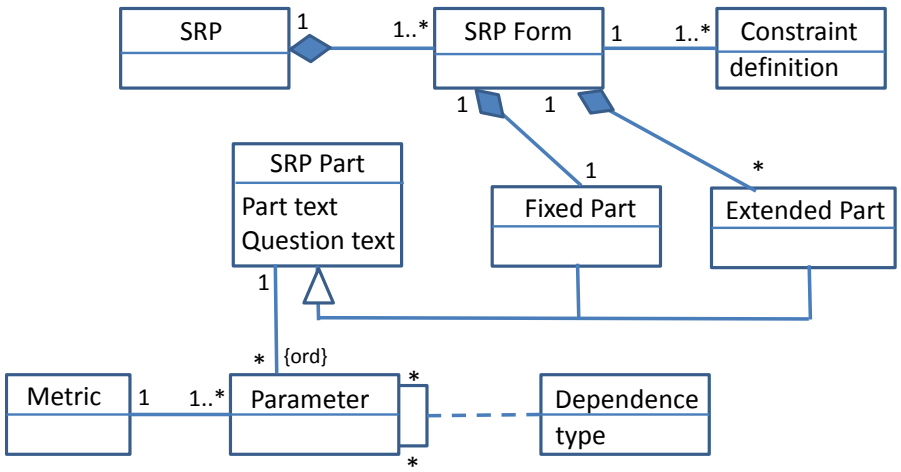


Figure 3.2. SRP structure

In this subsection the structure is described illustrated by the example.

3.3.1 SRP metadata

A SRP represents a specific problem that the client needs to solve and it is direcly related with the goal of the client. For each SRP, it is defined a set of attributes that provide metadata about it (see the attributes in the SRP class of the UML diagram). This set of attributes includes the name of the pattern, its description, the author, comments provided by the authors and users, its goal, the sources where the pattern has been taken from and some optional keywords to allow easier searches over the catalogue. In the SRP *Failure Alerts* the goal is *"To alert users about system failures"*. If a client wants that the system to develop has such alerts, this SRP will be applied.

3.3.2 Requirement forms

A goal of an SRP can be reached in different ways. Each of the possible solutions for a goal is expressed as a SRP form (see the class SRP Form in the UML diagram). In the case of the SRP *Failure Alerts* there are two possible forms: establish the specific alert types that are wished depending on the failure types (*Heterogeneous Failure Alerts*), or just establish the relevant alert types and failure types without any dependency among them (*Homogeneous Alert Types*).

Each form has a set of attributes that compose the metadata of the form (see the attributes in the UML class SRP Form): name of the form, description, author, comments, the modification date, the set of sources it comes from and finally each form has one fixed part and zero or more extended parts.

Each form has a fixed part, and several extended parts (see multiplicity among classes SRP Form and the classes Fixed Part and Extended Parts). The fixed part of a form is a text that explains the solution of a problem in a non-concrete way. The extended part of a form gives some extra information or restrictions about how to solve the goal of the pattern in a more specific way.

3.3.3 SRP part

Every fixed and extended part has a common structure (see class SRP part in the UML).

The form text of a part (see the corresponding attribute in the class SRP part) is expressed as a sentence in natural language that may include some parameters that point to variables that can change along different projects (see multiplicity regarding the Parameter class in the UML). Example: “The system shall trigger %alerts% alerts in case of %failures% system failures”

The question text (see the corresponding attribute in the class SRP part) is also expressed as a sentence in natural language that is related with the question that one would ask when he wants to know whether a product fulfils or not the requirement expressed in the fixed or extended part. Example: “The system shall trigger %alerts% alerts in case of %failures% system failures”.

When an SRP is selected and a SRP form is applied in a project, all the parameters established in the form text and the question text must be configured and replaced by a value. In order to define the values than a parameter can take, each parameter will be associated to a metric and optionally also will have a correctness condition. Metrics can be values from a specific domain, integer numbers, real numbers, boolean values, time values or set of values from the previous types. Examples: “failures: is a non-empty set of failure types. FailureTypes: FailureTypes = Set(FailureType). FailureType = Domain

(network crash, database crash, ...)" and "alerts: is a non-empty set of alert types. AlertTypes: AlertTypes = Set(AlertType). AlertType = Domain (Bip, Mail, SMS, ...)"

Requirement Pattern *Failure Alerts*	**Description**	This pattern expresses the need of having the system functionality to inform users about system failures at the moment the failure occurs.	
	Comments	----	
	Pattern goal	Alert users about system failures.	
	Author	GESSI-SSI	
	Sources	Requirement books from SSI Specialized literature	
	Keywords	Alert, crash response, Failure, System failure, technical support	
	Dependencies	----	
Requirement Form *Heterogeneous Failure Alerts*	**Description**	This form establishes a dependency among the type of alert and the type of system failure that occurs. The extension establishes which alerts are issued by which system failures	
	Comments	Application of extensions: Alerts for Failure Type: may (usually will) be applied more than once	
	Version date	2009-01-28 00:00:00.0	
	Author	GESSI-SSI	
	Sources	Requirement books from SSI Specialized literature	
	Fixed Part	**Question text**	Does the system trigger different types of alerts depending on the type of system failure?
		Form text	The system shall trigger different types of alerts depending on the type of system failure.
	Extended Part *Alerts for Failure Type*	**Question text**	Does the system trigger %alerts% alerts in case of %failures% system failures?
		Form text	The system shall trigger %alerts% alerts in case of %failures% system failures
		Parameter	**Metric**
		failures: is a non-empty set of failure types	FailureTypes: FailureTypes = Set(FailureType) FailureType = Domain (network crash, database crash, ...)
		alerts: is a non-empty set of alert types	AlertTypes: AlertTypes = Set(AlertType) AlertType = Domain (Bip, Mail, SMS, ...)
Requirement Form *Homogeneous Failure Alerts*	**Description**	This form does not establish any relationship among the type of alert and the type of system failure. It has extensions to determine the type of system failures and alerts in the system	
	Comments	Application of extensions: Alert Types, Failure Types: may be applied at most once each.	
	Version date	2009-03-20 00:00:00.0	
	Author	GESSI-SSI	
	Sources	Requirement books from SSI Specialized literature	
	Fixed Part	**Question text**	Does the system trigger an alert in case of system failure?
		Form text	The system shall trigger an alert in case of system failure
	Extended Part *Alert Types*	**Question text**	Does the system trigger %alerts% in case of system failure?
		Form text	The system shall trigger %alerts% alerts in case of system failure.
		Parameter	**Metric**
		alerts: is a non-empty set of alert types	AlertTypes: AlertTypes = Set(AlertType) AlertType = Domain (Bip, Mail, SMS, ...)
	Extended Part *Failure Types*	**Question text**	Does the system trigger alerts in case of %failures% failures?
		Form text	The system shall trigger alerts in case of %failures% failures.
		Parameter	**Metric**
		failures: is a non-empty set of failure types	FailureTypes: FailureTypes = Set(FailureType) FailureType = Domain (network crash, database crash, ...)

Figure 3.3. SRP example

3.3.4 Dependencies

Since requirement patterns are not isolated, different kind of dependencies have been identified between patterns.

Pattern dependencies extend the idea of having dependencies between requirements. These dependencies can be used during the elicitation process in order to help to sort the patterns and also can be propagated to the requirement specification process to improve their traceability if the pattern catalogue specifies that reaching a requirement implies reaching or avoid reaching another.

3.4 PABRE tools analysis

In this section the current architecture and implementation of the PABRE system tools is analysed since it may affect to the current project.

3.4.1 Three-tier architecture

Nowadays, in the software architecture design multitier architectures are usually used. In this kind of architectures, a simple objective is assigned to each tier, this allows to design scalable architectures (easily extendable in case of an increase of needs).

The main advantage of this architectural style is that the development can be made in different levels and, in case of having to implement any change; it is necessary just to modify the required level not having to review all the code of the different levels.

In addition, it allows to distribute the workload of the tool development in levels, in this way, each work group is not responsible of the rest of the levels and only has to know the available API between the levels.

The most widespread use of multi-tier architecture is the three-tier architecture (see Figure 3.4).

In the PABRE system, the three existent tools (PABRE-Man, PABRE-Proj and PABRE-Proj-Web) use a three-tier architectural model.

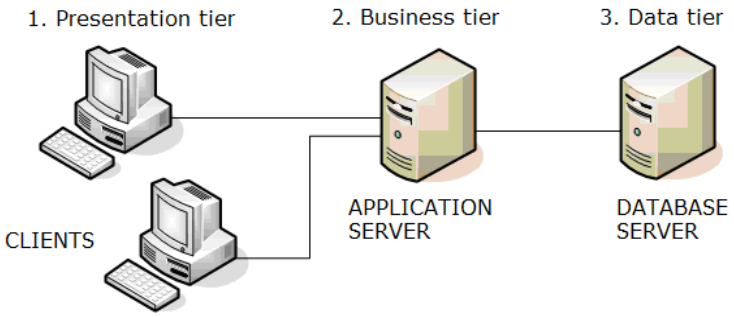


Figure 3.4. Three-tier architectural model

In software engineering a three-tier architecture is a client–server architecture in which presentation, processing, and data management functions are logically separated. A three-tier architecture is typically composed of a presentation tier, a business or data access tier, and a data tier.

Three-tier architecture has the following three tiers:

- Presentation tier

  This is the application nearest level to the user. The presentation tier displays information to the user, communicates the information and catches the inputs of the user. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. (In simple terms it's a layer which users can access directly such as a web page, or an operating systems GUI). This tier only communicates with lower layers through business layer. It is also known as GUI and one of their main features is that it should be friendly, understandable and usable.

- Business tier

  Also named application tier, business logic, logic tier, data access tier, or middle tier. It controls applications' functionality by performing detailed processing.

  It is where is located the code and algorithms that are executed when are processed the requests of the users and the responses are returned after this processing. In this tier is where all the rules that must be fulfilled in the domain are located. This tier communicates with presentation tier in order to receive the requests of the users and show the results and with the data tier in order to send

requests to database management system to store and retrieve persistent information from it.

- Data tier

  This tier consists of database servers. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

3.4.2 Used technologies

The three PABRE current tools (PABRE-Man, PABRE-Proj and PABRE-Proj-Web) were developed in the object oriented programming language Java and Hibernate was used as the framework for mapping the object-oriented domain model to the relational database.

Regarding the persistence of the information, Apache Derby was used as relational database management system.

Finally Google Web Toolkit (GWT) was used in the web version of PABRE-Proj-Web as the framework to create the JavaScript front-end of this application.

3.4.2.1 Java

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language.

Java was developed by Sun Microsystems at the beginning of the 90s. Among the goals they wanted to achieve with this language the object orientation and the portability can be emphasized.

The portability goal means that computer programs written in the Java language must run similarly on any hardware and operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode are intended to be interpreted by a virtual machine (VM) written specifically for the host hardware (see Figure 3.5). End-users commonly use a Java Runtime Environment (JRE)

installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

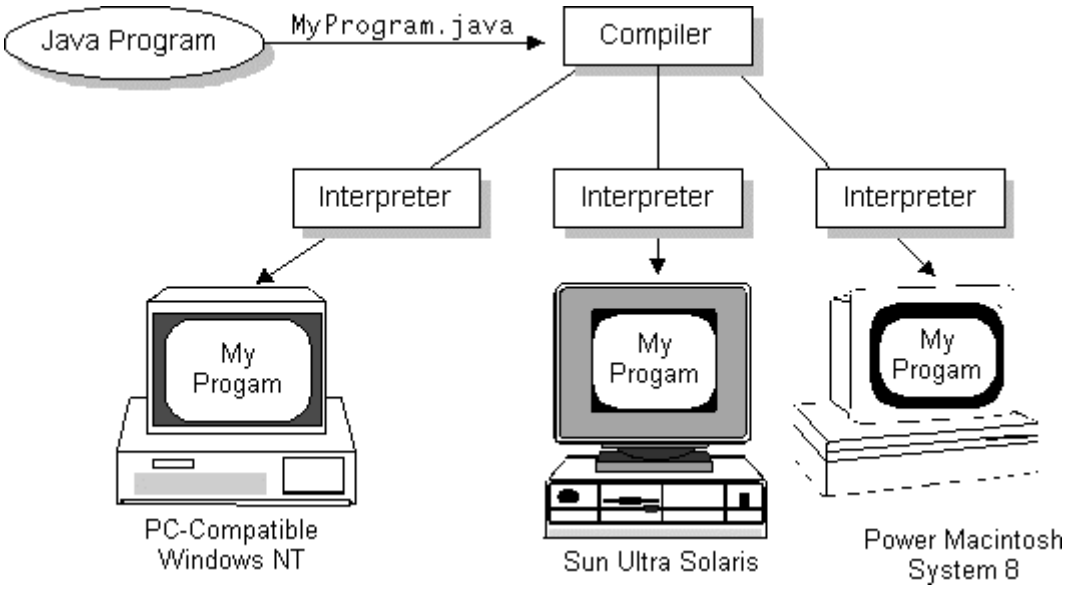


Figure 3.5. Java portability.

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable memory becomes eligible to be freed automatically by the garbage collector. Something similar to a memory leak may still occur if a programmer's code holds a reference to an object that is no longer needed

This language has a wide set of standardized libraries that provide a generic way to access host-specific features such as collections, graphics, threading, and networking.

Due to these and some others features made this language have a great acceptation and grow very fast in the developer's community. Thanks to this fact, the variety of available libraries for Java to support any kind of task is very wide and continually growing.

According to the PFC of Cristina Palomares (Palomares, 2010), who developed the second version of the PABRE-Man and PABRE-Proj tools, these are the features that where considered to take the decision of using Java in the developing of the applications:

- High level object oriented programming language

- High availability of programming interfaces (APIs) that offers support to developers in a wide variety of tasks.
- Complete portability on any hardware/operating-system platform.
- Low performance during execution that is not a problem due to the kind of tasks and algorithms that the application will perform.

On May 8, 2007, Sun finished the process, making all of Java's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright (Martens, 2007).

3.4.2.2 Hibernate

Hibernate is nowadays one of the most popular Java APIs. It is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.

In the context of object-oriented domain model and data persistence developers has to work on one hand with domain classes and on the other hand with relational database tables where these classes have to be stored and retrieved. Between this two models, ORMs are placed to isolate developers as much as possible from the details of this translation and Hibernate is the responsible of this task in these applications (see Figure 3.6).

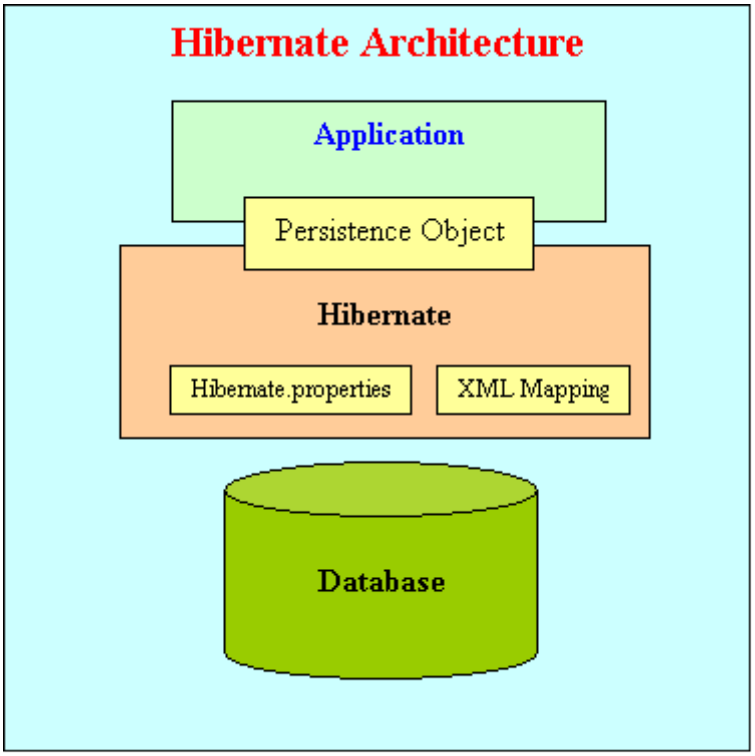


Figure 3.6. Hibernate Architecture

The mechanism Hibernate implements this translation providing a transparent persistence for Plain Old Java Objects (POJOs). Hibernates automatizes the mapping from these POJOs to database tables (and from Java data types to SQL data types) and provides data query and retrieval facilities. It generates SQL calls and relieves the developer from manual result set handling and object conversion. And a very important feature in the context of this project is that applications using Hibernate are portable to supported SQL databases with little performance overhead.

Collections of data objects are typically stored in Java collection objects such as Set and List. Hibernate can be configured to lazy load associated collections. Lazy loading is the default as of Hibernate 3. Related objects can be configured to cascade operations from one to the other. For example, a parent Album object can be configured to cascade its save and/or delete operation to its child Track objects. This can reduce development time and ensure referential integrity.

Hibernate provides an SQL inspired language called Hibernate Query Language (HQL) which allows SQL-like queries to be written against Hibernate's data objects. Criteria Queries are provided as an object-oriented alternative to HQL. Criteria Query is used to modify the objects and provide the restriction for the objects.

Hibernate is free software that is distributed under the GNU Lesser General Public License.

3.4.2.3 Apache Derby

Apache Derby is a relational database management system (DBMS) developed by the Apache Software Foundation that can be embedded in Java programs and used for online transaction processing.

Derby is a very light database, It has a 2.6 MB disk-space footprint and It does not need any installation process neither in the server nor in the client side, these were the main points this DBMS was selected for PABRE tools.

This DBMS can be packaged including the schema and data files of the system to distribute very easily all the PABRE tools with the patterns catalogue and the information of existing projects to clients.

Apache Derby is developed as an open source project under the Apache 2.0 license.

3.5 Domain class diagrams

Taking into account the domain of this project and the design of the API in the context of the web service that is going to be developed, a previous task of study of the hierarchy of the classes related to SRP and its metrics need to be performed.

In this analysis the hierarchy of the classes related to project management can be skipped since the goal of the web service is to provide a way to retrieve information about the SRP, and the creation or adaptation of the new or existing tools to manage the projects is responsibility of the clients and they can create their own hierarchy to manage this is the way it fits better their tools and methodologies.

In this hierarchy we can see the different entities of the domain of PABRE framework to analyse them in order to know all the information that clients should be able to access and to design the different operations and entities the API will return.

These class diagrams (Figure 3.7 and Figure 3.8) will be used as a guide in the design of the different operations and structures the web service will offer to clients.

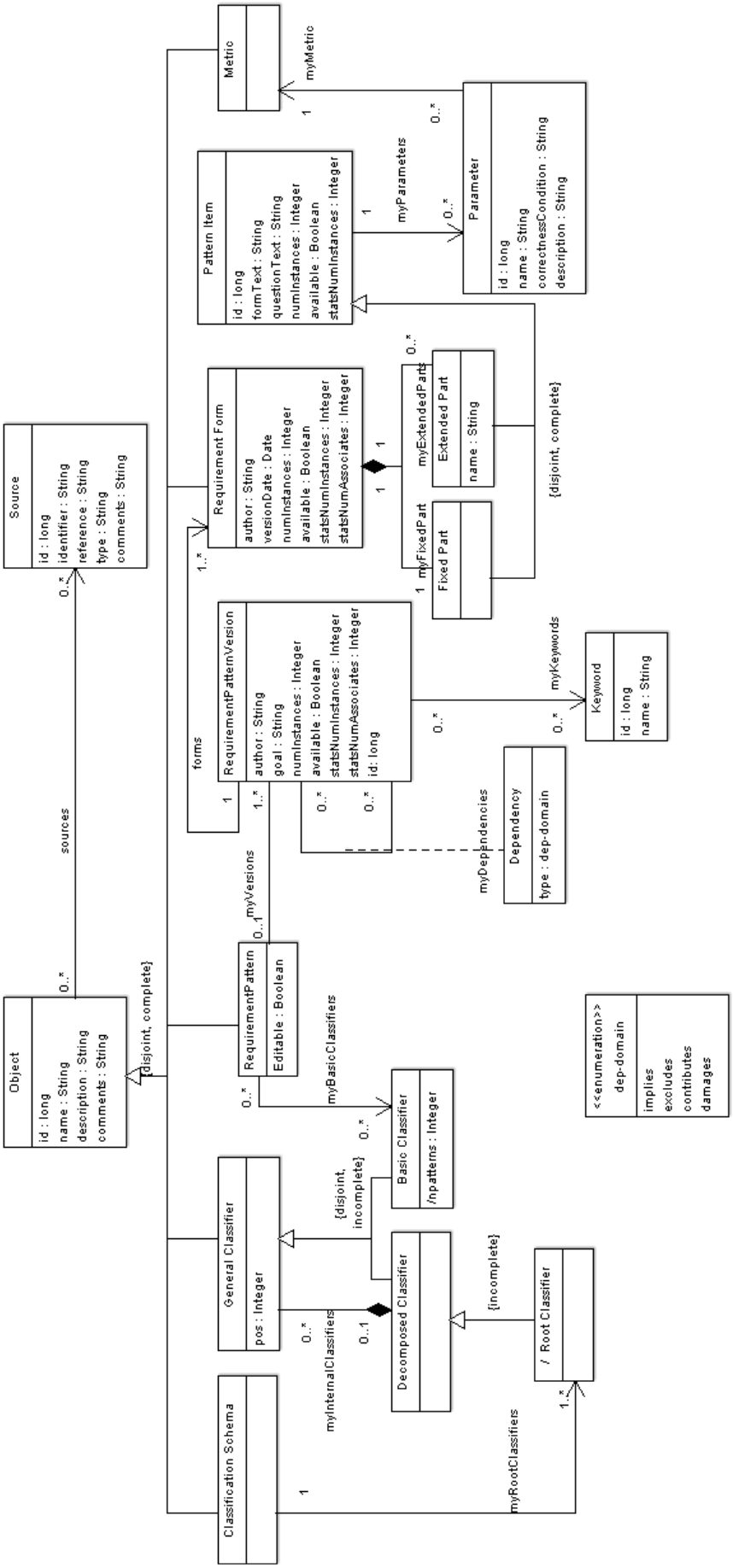


Figure 3.7. Class diagram of Requirement Patterns

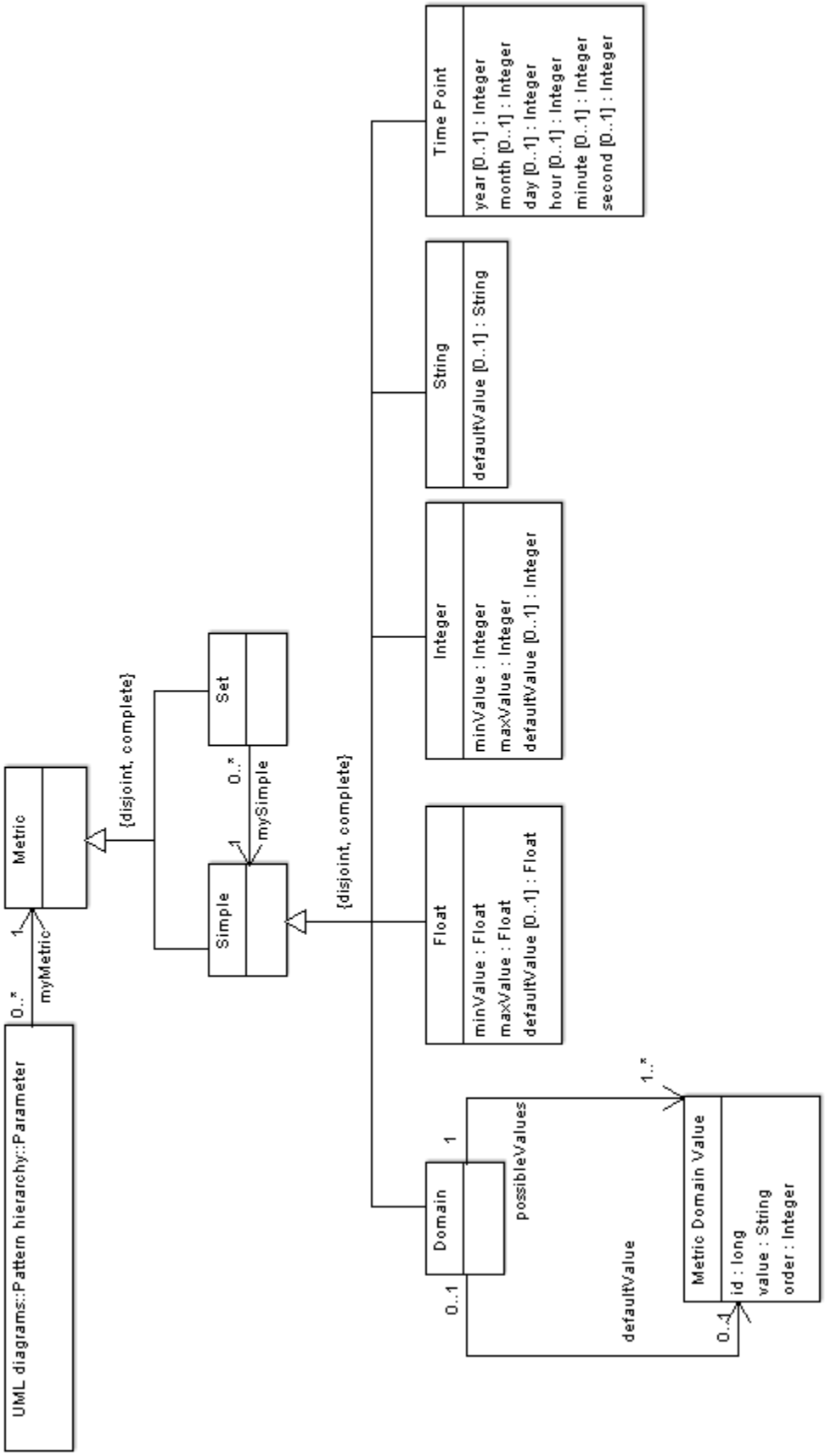


Figure 3.8. Class diagram of metrics

4 Analysis and design decisions

In this section it is detailed the requirements of the web services to develop classified depending on if they state the use that is going to be required of the services (functional requirements) or if they state needs about how they are going to be implemented (non-functional requirements). Then, it is explained the architectural decisions that have been taken to design the web service, the development platform and the used technologies, the design principles that have been followed during the design of the API and finally the design of the resources addressing, their representation and query parameters that have been considered necessary.

4.1 Requirements analysis

4.1.1 Functional requirements

- The web service must be compatible with the database schema of previous PABRE tools.
- The web service must allow consulting any information related to available classification schemas in the catalogue, their classifiers and their bound patterns.
- The web service must allow consulting any information related to available software requirement patterns in the catalogue, including their different versions, their forms, their parts, their dependencies, their keywords, their dependencies and their parameters and related metrics
- The web service must allow consulting any information related to available metrics in the catalogue, including their related metrics for set metrics.
- The web service must allow to receive an alphabetically ordered list of software requirement patterns by their names.
- The web service must allow to receive a list of unbound patterns for the available schemas.
- The web service must allow searching available patterns in the catalogue through their name and their keywords.

4.1.2 Non-functional requirements

- The web service should allow to receive requests and be compatible with the higher number of tools and programming languages possible in order to allow clients develop their tools using the widest variety of technologies and architectures possible.
- The web service must be platform independent, meaning that it must be able to be executed in any operating system, over any JavaEE application server.
- The web service must be compatible with an underlying MySQL DBMS.
- The web service must be compatible with an underlying Apache Derby DBMS.
- The web service must be able to be deployed in the current infrastructure of ESSI department (Tomcat 6.0 and MySQL 5.x).
- The web service must be able to have a permanent availability 24 hours per day.
- The web service must support concurrent multiple user requests.
- The web service must be able to establish a permanent connection with the DBMS.
- The web service must be able to process and respond any request with a maximum delay of 5 seconds for the current SRP catalogue.

4.2 Analysis of architectural decisions

This sections describes the two main architectural decisions of the web service. The decision of develop a SOAP based or REST based web service and the decision of using XML o JSON representation for the resources, explaining the motivations for selecting each one of the chosen alternatives.

4.2.1 RESTful web API

After analysing the different architectural and technological alternatives in section 2 (State of the art of web services) of this document. I have took the decision of make use of a RESTful approach for this project.

Analysing the application requirements and constraints described in the point 4.1 (Requirements analysis) I have conclude the following motivations for choosing this alternative:

- REST is very lightweight opposite to WS-* services that need a big infrastructure, It relies upon the HTTP standard to do its work, so the needed infrastructure to deploy this web services in lower and much more common, this will allow to deploy the web service will infrastructure that is already deployed in the client.

- This project doesn't need a complex and a strict API definition, situation where WS-* services and SOAP is most recommended.

- REST allow an easy way to version the API so that updates to the API do not break it for clients using old versions.

- REST is format-agnostic, meaning that they don't need to use XML such as SOAP. It can be used XML, JSON, HTML or whichever other format to code the messages, allowing choosing one or several format to adjust it to the format it fits better the needs of the client. Actually, you can also choose other format in SOAP either, but it hardly makes sense since it already is parsed to XML in the envelope.

- REST have less and more simple requirements relying on less message overhead because they do not have a SOAP envelope to wrap every call and response, this a big overload in the messages if we are not going to make any profit on the extra functionalities that SOAP protocol provides .

- REST methods and entities are more human readable, above all, in CRUD applications, so it should be easier for the web service clients to understand and create their own tools to access the pattern catalogue.

- REST provides a quicker access to external services, SOAP is much more structured and the average time to prepare all the required infrastructure and tools usually ends up in a higher average time to develop tools that use the service. It is true that there are a lot of libraries and frameworks that have made SOAP simpler and easier, but they abstract away a lot of redundancy making debugging and testing more complex.

- SOAP was designed for closed and controlled distributed computing environments while as REST was designed for a point to point open environments. The standard

architecture for the users of this web servicer is expected to be nearer to a point to point environment.

- WS-* services need more architectural decisions to be taken than for RESTful services, increasing the necessary knowledge about the alternatives and the risks of choosing a wrong one. Although it is true, that several of the decisions that are very easy to make for RESTful services lead to a later significant development efforts and technical risk, for example the design of the exact specification of the resources, their URI addressing scheme and resources relationship.

- RESTful services are typically used for tactical, ad hoc integration over the Web (Mashups), on the other hand, WS-* services are most used in professional enterprise application integration scenarios with a longer lifespan and advanced QoS requirements, that is not this case.

- REST causes less duplication because HTTP already represents operations like DELETE, PUT, GET, POST, etc. that have to otherwise be represented in a SOAP envelope.

- REST seems more standardized because HTTP operations are universal, well understood and operate consistently opposite to SOAP operations that have their own particular semantic and can become very particular for each web service.

- REST is easier to test because is human readable and testable with very simple tools, while is harder to test SOAP just with a browser.

4.2.2 JSON format

JSON (JavaScript Object Notation), is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

It has been decided to use JSON as an encoding format for messages of the web service over the XML alternative due to this application requirements and constraints and the nature of this web service (see comparison Figure 4.1).

http://localhost:8080/Json/SyncReply/Contacts

```
{
  - Contacts: [
      - {
            FirstName: "Demis",
            LastName: "Bellot",
            Email: "demis.bellot@gmail.com"
        },
      - {
            FirstName: "Steve",
            LastName: "Jobs",
            Email: "steve@apple.com"
        },
      - {
            FirstName: "Steve",
            LastName: "Ballmer",
            Email: "steve@microsoft.com"
        },
      - {
            FirstName: "Eric",
            LastName: "Schmidt",
            Email: "eric@google.com"
        },
      - {
            FirstName: "Larry",
            LastName: "Ellison",
            Email: "larry@oracle.com"
        }
    ]
}
```

http://localhost:8080/Xml/SyncReply/Contacts

```
<ContactsResponse xmlns:i="http://www.w3.org/20
  <Contacts>
    <Contact>
      <Email>demis.bellot@gmail.com</Email>
      <FirstName>Demis</FirstName>
      <LastName>Bellot</LastName>
    </Contact>
    <Contact>
      <Email>steve@apple.com</Email>
      <FirstName>Steve</FirstName>
      <LastName>Jobs</LastName>
    </Contact>
    <Contact>
      <Email>steve@microsoft.com</Email>
      <FirstName>Steve</FirstName>
      <LastName>Ballmer</LastName>
    </Contact>
    <Contact>
      <Email>eric@google.com</Email>
      <FirstName>Eric</FirstName>
      <LastName>Schmidt</LastName>
    </Contact>
    <Contact>
      <Email>larry@oracle.com</Email>
      <FirstName>Larry</FirstName>
      <LastName>Ellison</LastName>
    </Contact>
  </Contacts>
</ContactsResponse>
```

Figure 4.1. Difference between the same data encoded as JSON and XML (Locken, 2013)

After an analysis of both alternatives it has been detected a wide set of advantages of JSON over XML in this concrete project (Justin, 2010) and (Sporny, 2013):

- It is easier to write programs to process JSON, it has few optional features, it is human-legible and reasonably clear than XML, its design is formal and concise, JSON documents are easy to create, and it uses Unicode. JSON is fundamentally easier to work with than XML.

- JSON has many of the advantages of XML: it's straightforwardly usable over the Internet, supports a wide variety of applications.
- XML focuses on documents and JSON focuses on data. In this case we can consider the information contained in the web services messages is data.
- JSON is generally less verbose than XML: it is more limited in its use of data types. JSON possesses a very limited set of data types. It's essentially restricted to null, Booleans, numerics, strings, arrays, and dictionaries. It doesn't even have a Date data type. Restricting itself to primitive data types bring an important advantage: It makes JSON deeply and immediately interoperable with pretty much any programming language. XML defines namespaces for all attributes, but this project web service does not need namespaces for their data attributes. In this case all the messages of the API can be encoded using atomic values or lists or hashes of atomic values, so, in this sense, JSON should be enough.
- XML without also talking about document schemas, entities and DTDs, whereas JSON is largely schema-less and entity-free. These may not sound like big issues, but they creep into the APIs that you use to work with XML – adding unnecessary complexity along the way.
- For a JavaScript client in a web browser, JSON is a natural fit. The XML APIs in the browser are more complex and uncomfortable and the natural mapping from JavaScript objects to JSON eliminates the serialization issues that you have to afford with XML.
- XML is a far too complex solution for the scope of the web service of this project and the requirements it covers. XML is great in its other problems domain, when namespaces, well-formed structures, mixed content documents, etc. is needed, but in this case XML requires to use complex features that this web service does not need to be successful. It needs a simple data structures and a simple, compact, data exchange format.
- Interfacing with JSON web services is typically much easier as well, since JSON fits nicely into most programming languages via associative arrays. In fact, the list of languages on JSON's web is quite long (Introducing JSON, 2013).

- Most people is, nowadays, choosing JSON over XML when it comes to serializing and transmitting their data. It is becoming the API provider's choice (DuVander, 2011).

4.3 Development platform and technological decisions

This section describes the environment used in the context of this project for developing the web service, the test client interface and the database migration. It contains a brief description of each tool, the motivations for selecting them, the version that has been used and their distribution license.

4.3.1 Java

The decision of using Java as the programming language for developing the web services was mainly taken for compatibility reasons.

Using Java will allow reusing most of the classes of the code of Pabre-Man of the business layer saving a lot of developing time since most of the behaviour of the logic of the system is going to remain intact.

Also, the rest of reasons that caused Java to be the selected programming language described in the section 3.4.2.1 (Java) for previous tools already remain in the context of this project, and the technological infrastructure of the client is already able to support it through a Tomcat Server as a servlet container.

The version used in this project has been Java Platform, Enterprise Edition 7 (Java EE 7) with JDK 6 Update 45.

As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License.

4.3.2 Eclipse

Eclipse is a multi-language Integrated development environment (IDE) comprising a base workspace and an extensible plug-in system for customizing the environment. It is written mostly in Java. It can be used to develop applications in Java and, by means of various plug-ins, other programming languages.

The motivation for selecting Eclipse as the IDE for this project is mainly based on the previous experience and knowledge of this environment. I have used it a lot of time in my academic works and it fill the needs of this project.

The version used in this project has been Eclipse Java EE IDE for Web Developers version Juno Service Release 1.

Eclipse is released under the terms of the Eclipse Public License, Eclipse SDK is free and open source software (although it is incompatible with the GNU General Public License).

4.3.3 Apache Tomcat

Apache Tomcat is a web server and servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java Servlet and the Java Server Pages (JSP) specifications from Sun Microsystems, and provides a "pure Java" HTTP web server

environment for Java code to run in. Apache Tomcat is the world's most widely used web application server, with over one million downloads per month and over 70% penetration in the enterprise datacentre.

Apache Tomcat includes tools for configuration and management, but can also be configured by editing XML configuration files.

The standard deployment format for web applications is a .war file. Applications can be just located into the webapps folder and the next time tomcat starts it will unpack the war and make the application available.

The main motivation to select Tomcat as the servlet container is that the technological infrastructure of the client is already able to support it and the previous experience with this servlet container in past projects.

The version used in this project developing has been Apache Tomcat version 7.0.35, although the final deployment under the client infrastructure could be performed over a different version.

Apache Tomcat is an open source software developed in an open and participatory environment and released under the Apache License version 2.

4.3.4 Jersey

Java defines REST support via the Java Specification Request 311 (JSR). This specification is called JAX-RS (The Java API for RESTful Web Services). JAX-RS is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.

According to the Java EE 6 Tutorial, Volume 1: Jersey (Oracle, s.f.), Jersey is the Oracle reference implementation for this specification for JAX-RS. It implements support for the annotations defined in JSR 311, making it easy for developers to build RESTful web services by using the Java programming language.

Jersey allows developing RESTful Web services that seamlessly support exposing your data in a variety of representation media types and abstract away the low-level details of the client-server communication. Also, Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides its own API that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Jersey also exposes numerous extension SPIs so that developers may extend Jersey to best suit their needs.

Jersey uses a servlet which scans predefined classes to identify RESTful resources and via the web.xml configuration file of the web application, registers this servlet which is provided by the Jersey distribution.

This servlet analyses the incoming HTTP request and selects the correct class and method to respond to this request. This selection is based on annotations in the class and methods.

The version used in this project has been Jersey 1.17.

Jersey is dual licensed under 2 OSI approved licenses:

- COMMON DEVELOPMENT AND DISTRIBUTION LICENSE (CDDL - Version 1.1)
- GNU General Public License (GPL - Version 2, June 1991) with the [“Classpath Exception”]

4.3.5 Jackson JSON processor

Jersey uses serialization frameworks like JAXB (XML and JSON) and Jackson (JSON) for converting POJO objects to and from JSON. POJO mapping support represents the easiest way to convert Java Objects to JSON/XML and back. What this feature allows to do is deal with actual model classes in a service with no regard for their serialization and deserialization.

- Using Jackson library: This method represents the easiest way to convert Java Objects to JSON and back. To use this approach, you only need to turn the FEATURE_POJO_MAPPING feature on and all. This approach only support JSON mapping.
- Using JAXB (Java Architecture for XML Binding): This mapping easily produces/consumes both JSON and XML data format. Working with this model needs a previous configuration of the POJOs using annotations and they could be handled as simple Java beans at the same time you can easily produce/consume both JSON and XML data format (jasonray, 2013).

Taking into account that the design of the web service only considers the JSON serialization, it has been decided to use Jackson library. In any case, if in a future, the requirement of deal with XML media type for transactions were need, it should be easy to implement it changing the serialization method to JAXB and annotating the POJOs.

Jersey libraries include Jackson libraries version 1.9.2, but in the moment this project was started there was available a new version of Jackson 2.1.4 that. Since 2.x was a major versions upgrade, it is not plug-in backwards compatible with old code; instead, code using it must be recompiled. The reason for this is to both allow co-existence of Jackson 1.x and 2.x versions (FasterXML, 2013). But Jackson 2.x has some useful features that are interesting in the context of this project, specially the new @JsonFormat annotation that allows to configure the serialization format of dates.

This upgrade means that Jersey libraries related to Jackson must be replaced by Jackson 2.1.4 libraries, this is automatically done by Apache Maven once is configured in its "Project Object Model" file.

Jackson JSON processor is an Open Source product available under two OSS-approved licenses with users choosing which license they want to use: Apache License (AL) 2.0 or LGPL 2.1.

4.3.6 Hibernate

The decision of use Hibernate was mainly base on reusing on compatibility and code reusing reasons, since current tools of the PABRE framework are developed using this object-relational mapping (ORM) library.

This means that if the database schema is preserved, the xml mapping file of hibernate should be reusable to map the classes and entities of the web service business layer over the database.

Hibernate also will help a lot in the database migration of the persistent data. SQL (Structured Query Language) is the most extended language for managing data held in a relational database management systems, unfortunately, despite SQL being standardized since 1986, a lot of different implementations exist. They deviate more or less from each other, making developing applications that would work with a range of different SQL servers particularly difficult.

This means that changing the underlying database probably would result in a new long task of refactoring all SQL sentences adapting them to the dialect of the new database. But, by using hibernate, it creates a new intermediate layer where all queries are made in HQL (Hibernate Query Language) and it allow to configure a dialect property that makes Hibernate generate the appropriate SQL for the chosen database. So, changing this property, the connection parameters of the database and the database connector driver should be enough from the code refactoring point of view.

The Hibernate version used in this project has been 3.3.2, the same version used in Pabre-Man in order to avoid problems with the mapping and configuration used for that software that are going to be reused in this project.

Hibernate is Free Software licensed under the LGPL v2.1. The LGPL is sufficiently flexible to allow the use of Hibernate in both open source and commercial projects.

4.3.7 Apache Maven

Maven is a build automation tool used primarily for Java projects. Maven can be used to build and manage projects written in Java, C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.

Maven uses an XML file to describe the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging.

The main motivation to use Maven was its dependency management. Maven's dependency-handling mechanism is organized around a coordinate system identifying individual artifacts such as software libraries or modules.

Since this project needs, libraries such as Hibernate, Jersey, Jackson, database connector drivers, etc., Maven simplifies the management of this libraries. By using Maven a project simply has to declare the libraries dependencies in its POM (Project Object Model) file and Maven will automatically download the dependency and the dependencies that those libraries itself needs (called transitive dependencies) and store them in the user's local repository. Maven 2 Central Repository is used by default to search for libraries, but one can configure the repositories to be used (e.g., company-private repositories) within the POM.

One of the most important principles of the Maven build system is that there are standard locations for all of the files in the Maven project. There are several advantages to this principle. One advantage is that Maven projects normally have an identical directory layout, making it easy to find files in a project (once you are familiar with the standard layout). Another advantage is that the various tools integrated with Maven need almost no initial configuration. For example, the Java compiler knows that it should compile all of the source files under src/main/java and put the results into target/classes. In Maven documentation can be found the directory layout expected by Maven and the directory layout created by Maven (The Apache Software Foundation, 2013).

In order to provide Apache Maven support in the Eclipse IDE, making it easier to edit Maven's pom.xml, manage dependencies, run a build from the IDE and much more the Eclipse plugin “m2eclipse” version 1.3.1 has been used. This simplifies the consumption of Java artifacts either being hosted on open source repositories such as Maven Central, or in the in-house Maven repository (The Eclipse Foundation, 2013).

Also the Maven Integration for WTP plugin has been installed. Also known as m2e-wtp, it aims at providing a tight integration between Maven Integration for Eclipse (m2eclipse) and the Eclipse Web Tools Project (WTP). m2e-wtp provides a set of m2e connectors used for the configuration of Java EE projects in WTP, bringing Maven features to the Eclipse IDE and helping to convert Eclipse projects to Maven.

The Apache Maven version used in this project has been 3.0.5.

Apache Maven is Free Software licensed under The Apache Software License, Version 2.0.

4.3.8 Git

Git is a distributed version control and source code management (SCM) system. Initially designed and developed by Linus Torvalds for Linux kernel development in 2005, Git is reported to have 36% market adoption as of 2013.

Every Git working directory is a full-fledged repository with complete history and full version tracking capabilities, not dependent on network access or a central server.

The main motivation of using this tool was the necessity of a source code management tool for checking source code in and from a backup repository reducing the risk of losing code or overwriting changes. This tool also will provide a version control system that can manage files through the development lifecycle, keeping track of which changes were made, when they were made, and why.

Also the built-in GUI tools haven been used for committing (git-gui) and browsing (gitk).

For store the code as a repository an academic account in GitHub has been created. GitHub was the most popular open source code repository site for software development projects that use the Git revision control system.

The Git version used in this project has been 1.8.1.

Git is free software distributed under the terms of the GNU General Public License version 2.

4.3.9 Apiary.io

Apiary is on a web application that his goal is to help developers to build and use. Apiary hopes to be for APIs what GitHub did for coding.

Apiary provides hosted PaaS platform helping companies design and develop APIs faster, support their API customers. The core of the self-service solution is an API Blueprint, an efficient format for describing an API, aspiring to define the new standard for REST API development. Apiary.io starts with what they call the API Blueprint, which is a custom DSL (domain-specific language) allowing developers to quickly describe their APIs, and

from this blueprint apiary.io will then generate API documentation, a debugging proxy and bug reports.

The main motivation of using this application is to have a platform to write and publish the API documentation following the documentation standards and presenting it in a nice interface to the clients.

4.3.10 MySQL

MySQL is the world's most widely used open-source relational database management system (RDBMS) that runs as a server providing multi-user access to a number of databases. MySQL is a popular choice of database for use in web applications, and is a central component of the widely used LAMP open source web application software stack.

Like other SQL databases, MySQL does not currently comply with the full SQL standard for some of the implemented functionality, including foreign key references when using some storage engines other than the 'standard' InnoDB. This problem could affect in the data migration task from derby, but it will be fixed with other tools like DdlUtils and Hibernate dialect selection.

The official MySQL front-end tool, MySQL Workbench, has been use in this project to manage and browse the databases of the different tools.

The main motivation to use MySQL in this project is because it was a requirement of the client in order to adapt the existing PABRE tools and this new project to the current infrastructure the client already has deployed.

The MySQL server version used in this project has been 5.6.

The MySQL development project has made its source code available under the terms of the GNU General Public License, as well as under a variety of proprietary agreements.

4.3.11 SQuirreL SQL Client

The SQuirreL SQL Client is a database administration tool. It uses JDBC to allow users to explore and interact with databases via a JDBC driver. It provides an editor that offers code completion and syntax highlighting for standard SQL. It also provides a plugin architecture that allows plugins to modify most of the application's behaviour to provide database-specific functionality or features that are database-independent. As this desktop application is written entirely in Java with Swing UI components, it should run on any platform that has a JVM.

The main motivation to use this database client has been its use in the development of the previous PABRE tools, its ability to connect and manage different database engines (in this case Apache Derby and MySQL) and the existence of the DBCopy plugin (DBCopy Plugin Team, 2013) for this client that allows copying database objects (schema definition and data) from and to different database engines, been specifically compatible with Apache Derby and MySQL, very useful for the migration task, although this tool was finally discarded in favour of DdlUtils because it did not cover the migration needs.

The SQuirreL SQL Client server version used in this project has been 3.4.0.

SQuirreL SQL Client is free as open source software that is distributed under the GNU Lesser General Public License.

4.3.12 Apache DdlUtils

DdlUtils is a small, easy-to-use component for working with Database Definition (DDL) files. These are XML files that contain the definition of a database schema, e.g. tables and columns. These files can be fed into DdlUtils via its Ant task or programmatically in order to create the corresponding database or alter it so that it corresponds to the DDL. Likewise, DdlUtils can generate a DDL file for an existing database.

DdlUtils strives to be database independent. DdlUtils uses the Turbine XML format, which is shared by Torque and OJB. This format expresses the database schema in a database-independent way by using JDBC datatypes instead of raw SQL datatypes which are inherently database specific.

In short, DdlUtils is both a library and a set of Ant tasks that allows the manipulation of schemas in a database. It allows to create and drop complete databases, and the creation, alteration and removal of tables. Additionally, it provides an easy way to insert data that is specified in XML, into a database and the reverse, extracting data from a database into an XML file.

This features will allow to get the schema and the database information in database engine independent XML files and store them in another different database engine accomplishing the database engine migration goal.

The Apache DdlUtils version used in this project has been 1.0.

Apache DdlUtils is an open source software distributed under the Apache License, Version 2.0.

4.4 API design principles

In this section it is described how the resources are going to be exposed in the web service and how they are going to be represented.

Resources are a fundamental concept in any RESTful API. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it.

Resources can be grouped into collections. Each collection is homogeneous so that it contains only one type of resource, and unordered. Resources can also exist outside any collection. In this case, we refer to these resources as singleton resources. Collections are themselves resources as well.

Collections can exist globally, at the top level of an API, but can also be contained inside a single resource. In the latter case, we refer to these collections as sub-collections. Sub-collections are usually used to express some kind of “contained in” relationship. (Figure 4.2)

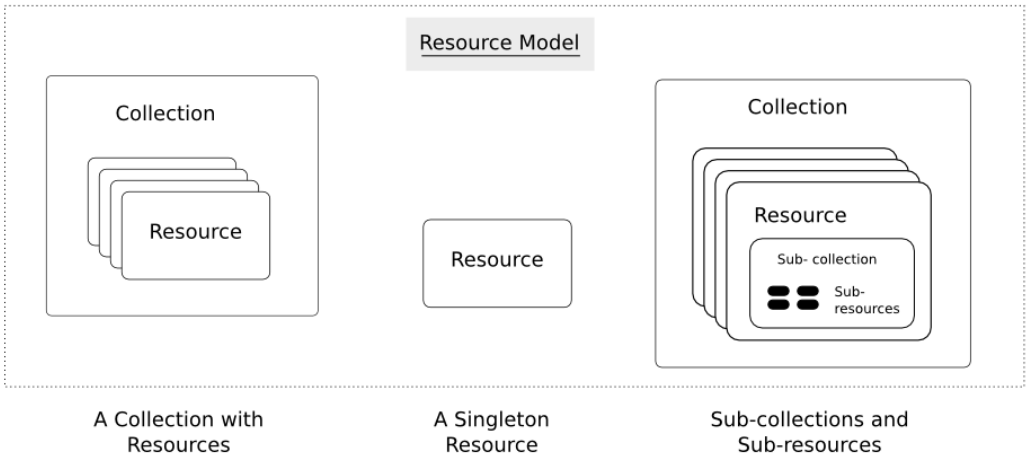


Figure 4.2. Key concepts in a RESTful API

An important thing to think about while designing resource representations is how to define the relationships between different resources. Doing so will allow consumers to navigate the “Web” of resources exposed by the service, and even discover how to use navigate service.

Taking into account that RESTful APIs do not have any official standard design practice for their resources representation the API should be designed thinking in the use cases of the clients of the API.

There are two ways to expose a resource to a client:

- Give the full information about a resource the client wants to retrieve including the information of the subcollections and subresources in a unique response, saving the client the necessity of make new requests in order to get the rest of the information.
- Give the client only the information that directly belongs to the resource and the URIs where the client can find the subcollections and subresources related to the requested resource, in order to allow the client to make new requests for them in they want to access that information, making some extra requests if they wanted the full information.

But this is impossible since the goal of the API is to expose the content of the requirement patterns catalogue in order to provide clients the information they need to create or adapt their tools designing their own use cases. So, the main design principle in the design of the resources addressing and representation is trying to find an appropriate balance between the number of requests and the size of the response in order to get the desired information.

4.5 Resources addressing schema

One of the most important decisions in the API design is the URI addressing schema of the different resources. The resource path section of a URI identifies the resource to be interacted with and must enable any aspect of the data model exposed by the web service to be addressed.

As it was previously explained in section 4.4 (API design) the URIs of the addressing schema are divided in the two groups:

- Collections: Return a list of available resources in the system and are represented in Figure 4.3 (Resources addressing schema) using a folder icon.

- Individual resources: Returns information about a single resources identified by its unique id in the URI and are represented in Figure 4.3 (Resources addressing schema) using a file icon.

The resources that haven been decided to be exposed in the addressing schema are:

- Patterns collection: This resource will return a list of all the patterns of the system ordered alphabetically. This list will have a limited set of properties of each pattern and the URI of each individual pattern to access the rest of the information.

- Individual Pattern: They represent an individual requirement pattern, when this resources is requested all the information about it must be returned, or in other case, at least, the URI to access the information related to this resource, such as sources, their forms, their versions, their keywords, their metrics, etc. When a specific pattern is requested the system will automatically return the last version of that pattern.

- Pattern versions collection: This resource will return a list of all the versions of a specific requirement pattern ordered by date. This list will have a limited set of properties of each version and the URI of each individual version to access the rest of the information.

- Individual Pattern version: They represent a specific version of a pattern, this resource will return the same information as a pattern for a concrete version, including the URI to access the pattern it belongs to.

- Metrics collection: This resource will return a list of all the metrics of the system. This list will have a limited set of properties of each metric and the URI of each individual metric to access the rest of the information.

- Individual Metric: This resource represent an individual metric, when this resource is requested all the information about that metric must be returned.

- Sources collection: This resource will return a list of all the sources of the system. This list will have a limited set of properties of each source and the URI of each individual source to access the rest of the information.

- Individual Source: This resource represent an individual source, when this resource is requested all the information about that source must be returned.

- Schemas collection: This resource will return a list of all the schemas of the system. This list will have a limited set of properties of each schema and the URI of each individual schema to access the rest of the information.

- Individual Schema: This resource represent an individual schema, when this resource is requested all the information about that schema must be returned. It includes the full hierarchy of all their contained classifiers and patterns and the list of unbinded patterns of the schema

- Classifiers collection: This resource will return a tree with the hierarchy of the classifiers of a specific schema. This tree will have a limited set of properties of each classifier and the URI of each individual classifier to access the rest of the information.

- Individual Classifier: This resource represent an individual classifier in a schema, when this resource is requested all the information about that classifier must be returned. It includes the full hierarchy of all their contained sub classifiers and patterns.

Below, Figure 4.3 (Resources addressing schema), shows the designed addressing schema of the resources of Pabre-WS.

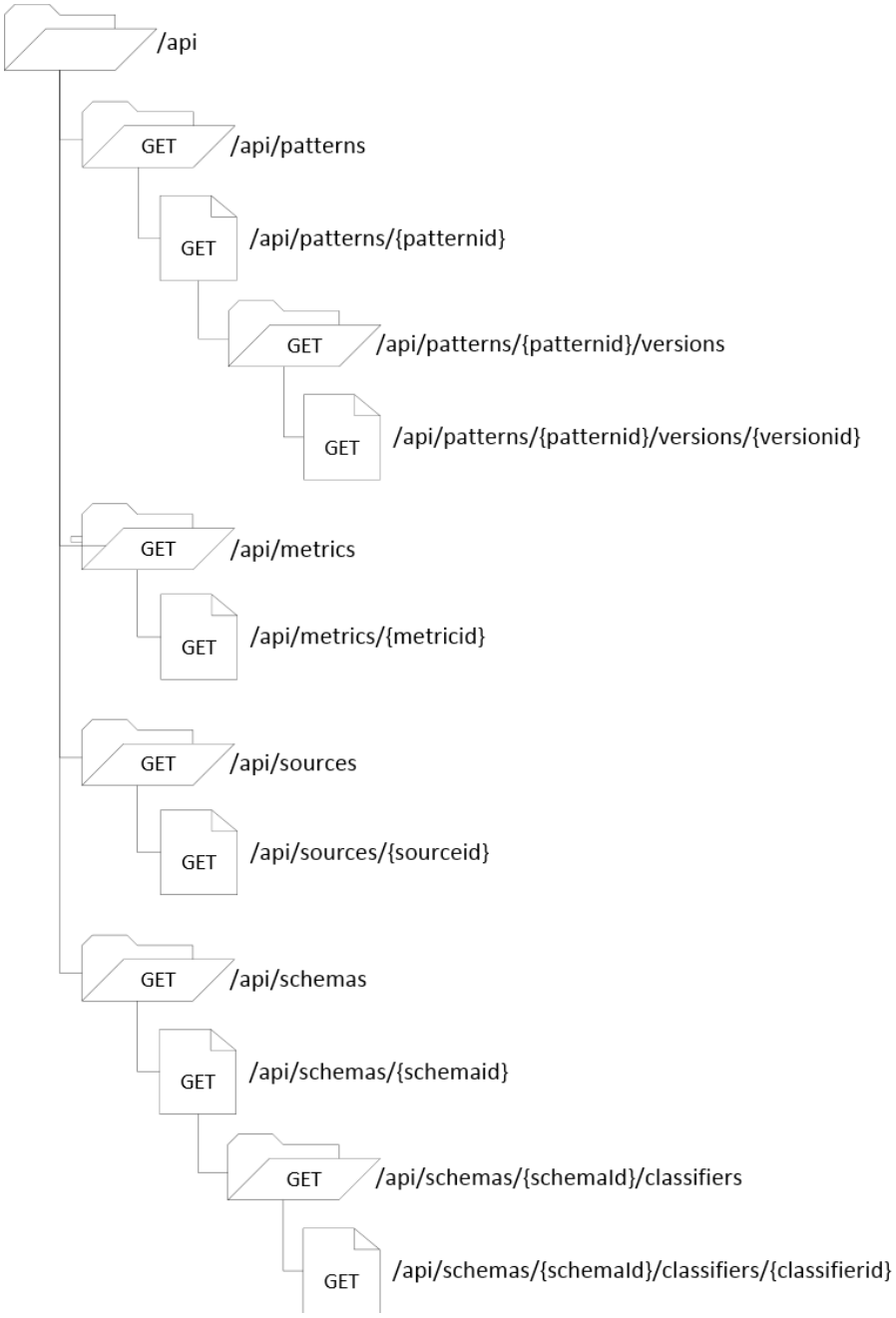


Figure 4.3. Resources addressing schema

4.6 Resources representation design

Resources have data associated with them. The richness of data that can be associated with a resource is part of the resource model for an API. It defines for example the available data types and their behaviour.

In JSON, just three types of data exist:

- scalar (number, string, boolean, null).

- array
- object

Scalar types have just a single value. Arrays contain an ordered list of values of arbitrary type. Objects consist of an unordered set of key:value pairs (also called attributes), where the key is a string and the value can have an arbitrary type.

In a RESTful API that agrees with the HATEOAS principle explained in section 2.2.1 (Architectural principles), in addition to exposing application data, resources also include other information that is specific to the RESTful API. Such information includes URLs and relationships. These URL identifies the address of the current resource and the addresses of resources that have some relationship for the current resource.

In order to find the balance point of requests number and response size explained in section 4.4 (API design) in this API two types of resource representation haven been designed:

- Complete: A representation that returns the full information of a resource including also the related individual resources and subcollections.
- Partial: An intermediate representation where only some relevant attributes of a resource are included and the URI location to request the resource and obtain the complete representation.

In the following sections the detailed design of each resource is explained.

4.6.1 Individual pattern

This resource is returned when `/api/pattern/{patterned}` URI is requested.

The individual patterns resource provides information about the requirement pattern and its last version information. It is represented as a JSON object that is a complete representations of a requirement pattern, their subresources and their subcollections.

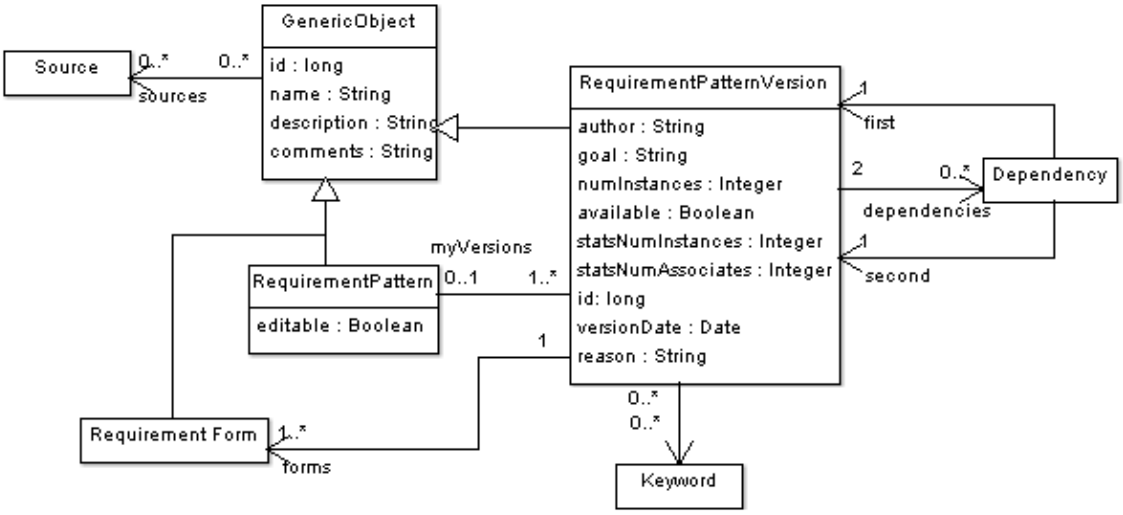


Figure 4.4. RequirementPattern bussiness class and relationships.

This complete representation of an individual pattern is based on the Requirement Pattern Business class (Figure 4.4) and its last RequirementPatternVersion object associated and it contains the fields described below.

All these fields are obtained from the RequirementPattern object with the specified {patternid}:

- name: References the field name.
- description: References the field description.
- comments: References the field comments.
- sources: Array with a partial representation of the sources referenced in the field "sources" described in section 4.6.8 (Sources collection).
- editable: References the field editable.
- versions: Array with a partial representation of the Requirement Pattern Versions referenced in the field myVersions described in section 4.6.4 (Pattern versions collection).

All these fields are obtained from the last version of the RequirementPatternVersion object related to the RequirementPattern object with the specified {patternid}:

- author: References the field author.
- available: References the field available.

- dependencies: Array with a complete representation of the dependencies referenced in the field dependencies. Section 4.6.1.1 (Dependency complete representation).
- forms: Array with a complete representation of the Requirement Forms referenced in the field forms. Section 4.6.1.3 (Requirement Form complete representation).
- goal: References the field goal.
- keywords: Array with the partial representation of the keywords referenced in the field keywords. Section 4.6.1.2 (Keyword partial representation).
- numInstances: References the field numInstances.
- statsNumAssociates: References the field statsNumAssociates.
- statsNumInstances: References the field statsNumInstances.
- versionDate: String that represents the Date referenced in the field versionDate.

All these fields are computed:

- uri: String that references the URI of this individual pattern.
- versionUri: String that references the URI of this individual pattern version.

The JSON schema of a complete representation of a Requirement Pattern is shown in Figure 4.5.

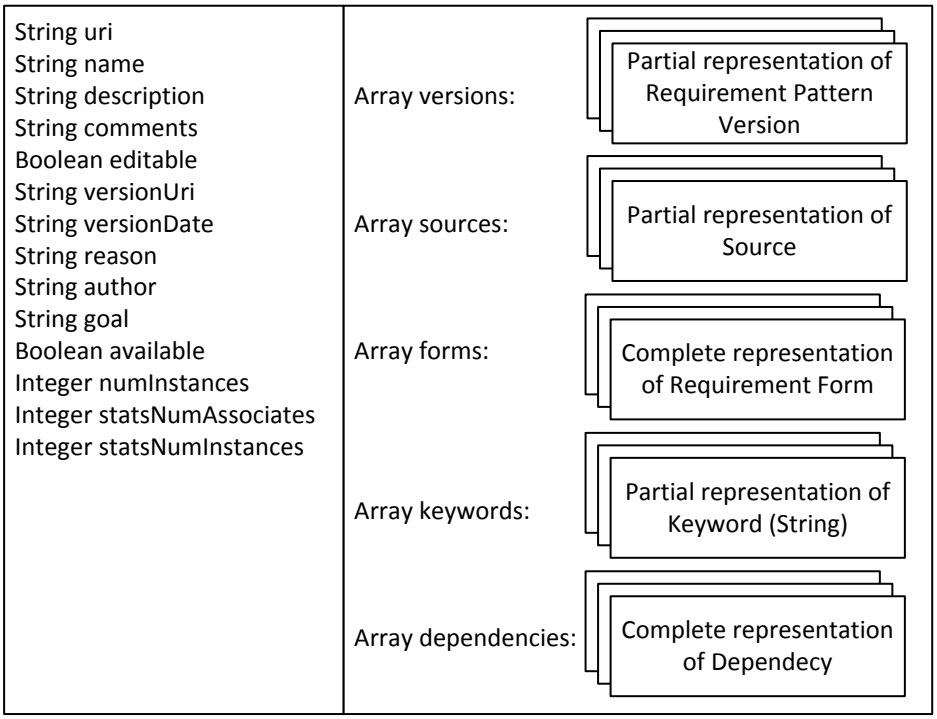


Figure 4.5. Complete representation of individual requirement pattern.

4.6.1.1 Dependency complete representation

This complete representation contains the following fields from the Dependency (Figure 4.6) objects referenced in "dependencies" field:

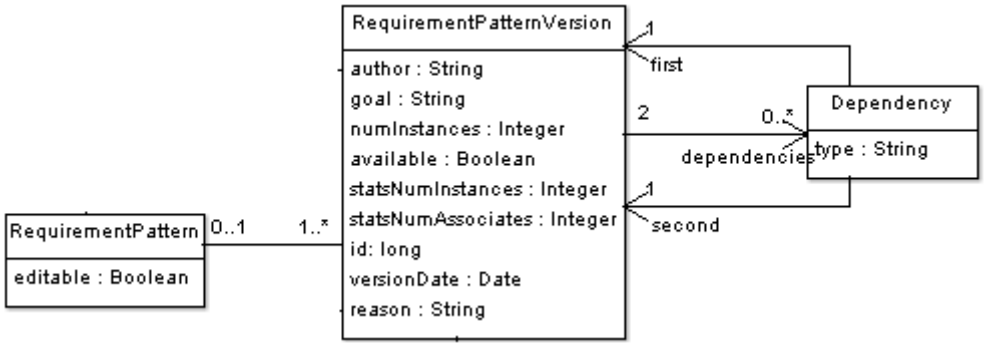


Figure 4.6. Dependency business class and relationships.

- first: Partial representation of the RequirementPatternVersion referenced in the field first. This partial representation contains the following fields from the RequirementPatternVersion object referenced in the "first" field:
    - uri: String that references the URI of this individual pattern version
    - requirementPattern: Partial representation of the RequirementPattern referenced in the field myRequirementPattern. This partial representation contains the following fields from the

RequirementPattern object referenced in the “myRequirementPattern” field:

- name: References the field name.
- uri: String that references the URI of this individual pattern.

- second: Partial representation of the RequirementPatternVersion referenced in the field second. This partial representation contains the following fields from the RequirementPatternVersion object referenced in the “second” field:
  - uri: String that references the URI of this individual pattern version
  - requirementPattern: Partial representation of the RequirementPattern referenced in the field myRequirementPattern. This partial representation contains the following fields from the RequirementPattern object referenced in the “myRequirementPattern” field:
    - name: References the field name.
    - uri: String that references the URI of this individual pattern.
- type: References the field type.

The JSON schema of a complete representation of a Dependency is shown in Figure 4.7.

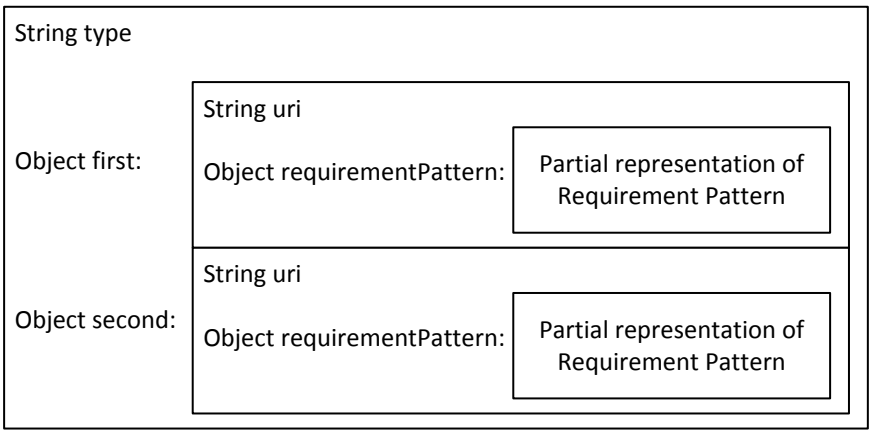


Figure 4.7. Dependency complete representation

4.6.1.2 Keyword partial representation

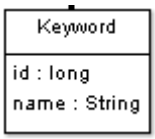


Figure 4.8. Keyword business class.

This partial representation is a simple String that references the following field of Keyword (Figure 4.8) objects referenced in “keyword” field:

- name

The JSON schema of a complete representation of a Keyword is shown in Figure 4.9.

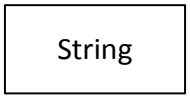


Figure 4.9. Partial representation of Keyword

4.6.1.3 Requirement Form complete representation

This complete representation is based in RequirementForm business class (Figure 4.10).

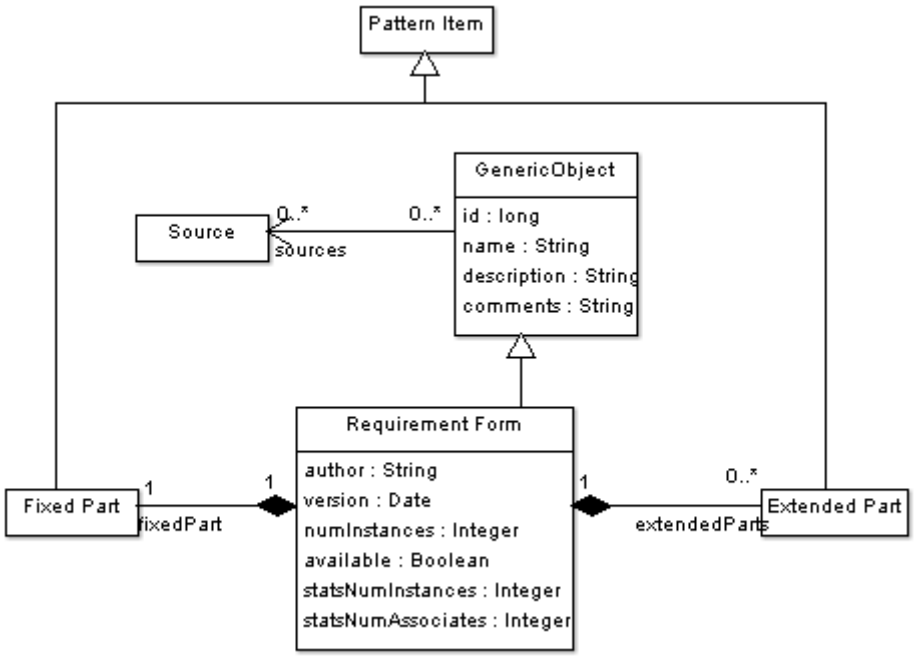


Figure 4.10. RequirementForm business class and relationship

It contains the following fields from the RequirementForm objects referenced in “forms” field:

- name: References the field name.
- description: References the field description.
- comments: References the field comments.
- sources: Array with a partial representation of the sources referenced in the field “sources” described in section 4.6.8 (Sources collection).

- author: References the field author.
- modificationDate: String that represents the Date referenced in the field version.
- available: References the field available.
- numInstances: References the field numInstances.
- statsNumAssociates: References the field statsNumAssociates.
- statsNumInstances: References the field statsNumInstances.
- fixedPart: Complete representation of the FixedPart object referenced in the field fixedPart. Section 4.6.1.4 (Fixed Part complete representation).
- extendedParts: Array with a complete representation of the ExtendedPart objects referenced in the field extendedParts. Section 4.6.1.5 (Extended Part complete representation).

The JSON schema of a complete representation of a Requirement Form is shown in Figure 4.11.

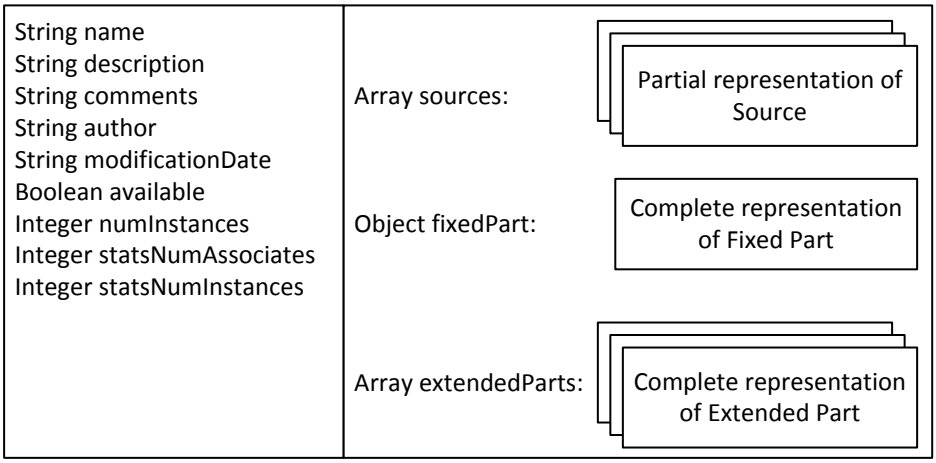


Figure 4.11. Complete representation of Requirement Form

4.6.1.4 Fixed Part complete representation

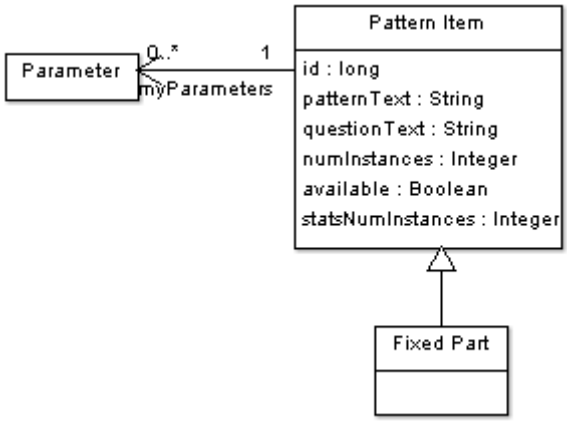


Figure 4.12. FixedPart business class and relationships

This complete representation contains the following fields from the FixedPart objects referenced in "fixedPart" field:

- formText: References the field patternText.
- questionText: References the field questionText.
- numInstances: References the field numInstances.
- available: References the field available.
- statsNumInstances: References the field statsNumInstances.
- parameters: Array with a complete representation of the Parameter objects referenced in the field myParameters. Section 4.6.1.6 (Parameter complete representation).

The JSON schema of a complete representation of a Fixed Party is shown in Figure 4.13.

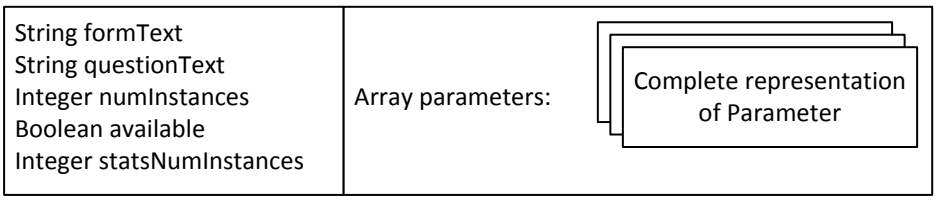


Figure 4.13. Complete representation of Fixed Part

4.6.1.5 Extended Part complete representation

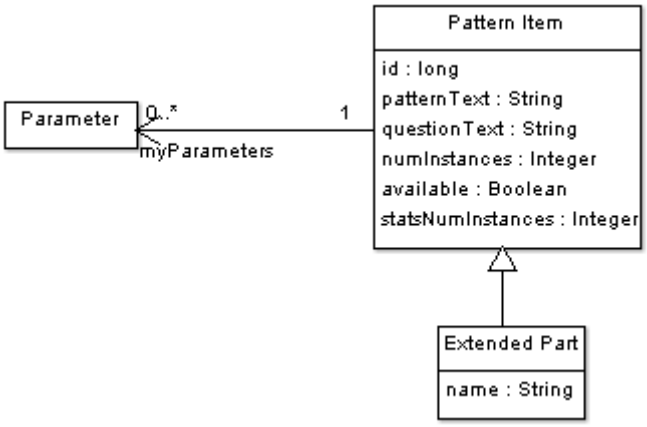


Figure 4.14. ExtendedPart business class and relationships

This complete representation contains the following fields from the FixedPart objects referenced in "fixedPart" field:

- name: References the field name.
- formText: References the field patternText.
- questionText: References the field questionText.
- numInstances: References the field numInstances.
- available: References the field available.
- statsNumInstances: References the field statsNumInstances.
- parameters: Array with a complete representation of the Parameter objects referenced in the field myParameters. Section 4.6.1.6 (Parameter complete representation).

The JSON schema of a complete representation of an Extended Part is shown in Figure 4.15.

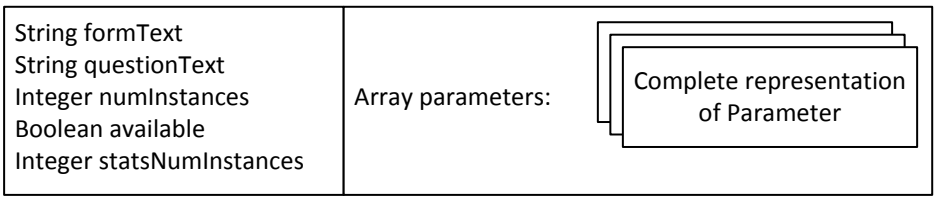


Figure 4.15. Complete representation of Extended Part

4.6.1.6 Parameter complete representation

This complete representation of a parameter is based in Parameter business class (Figure 4.16).

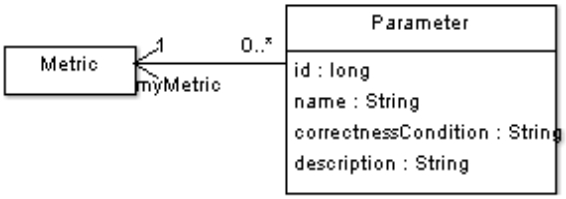


Figure 4.16. Parameter business class and relationships

It contains the following fields from the FixedPart objects referenced in "fixedPart" field:

- name: References the field name.
- correctnessCondition: References the field correctnessCondition.
- description: References the field description.
- metric: Complete representation of the Metric object referenced in the field myMetric described in section 4.6.5 (Individual Metric)

The JSON schema of a complete representation of a Parameter is shown in Figure 4.17.

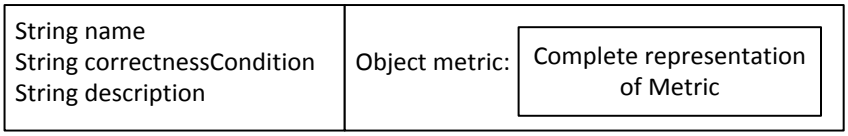


Figure 4.17. Complete representation of Parameter

4.6.2 Patterns collection

This resource is returned when /api/patterns URI is requested.

The patterns collection resource provides a list of patterns that can or not be filtered using some parameters. It is represented as an array of partial representation of requirement patterns.

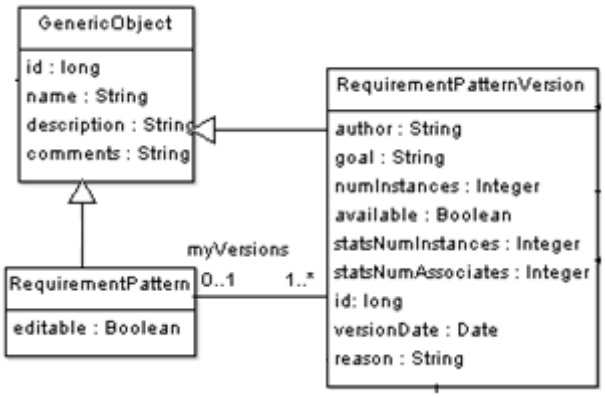


Figure 4.18. RequirementPattern business class and relationships

This partial representation of a requirement patterns is based on the Requirement Pattern Business class (Figure 4.18) and it contains the fields described below.

These fields are obtained from the RequirementPattern objects:

- name: References the field name.
- editable: References the field editable.

This fields is obtained from the last version of the RequirementPatternVersion object related to the RequirementPattern objects:

- available: References the field available.

This field is computed:

- uri: String that references the URI of the individual pattern

The JSON schema of a partial representation of a Requirement Pattern is shown in Figure 4.19.

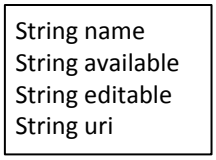


Figure 4.19. Partial representation of Requirement Pattern

4.6.3 Individual pattern version

This resource is returned when `/api/pattern/{patternid}/version/{versioned}` URI is requested.

The individual pattern version resource provides information about a specific version of a requirement pattern and its requirement version. It is represented as a JSON object that is a complete representations of requirement pattern version, their subresources and their subcollections.

This complete representation of an individual pattern is based on the Requirement Pattern Business class (Figure 4.20) and it contains the fields described below.

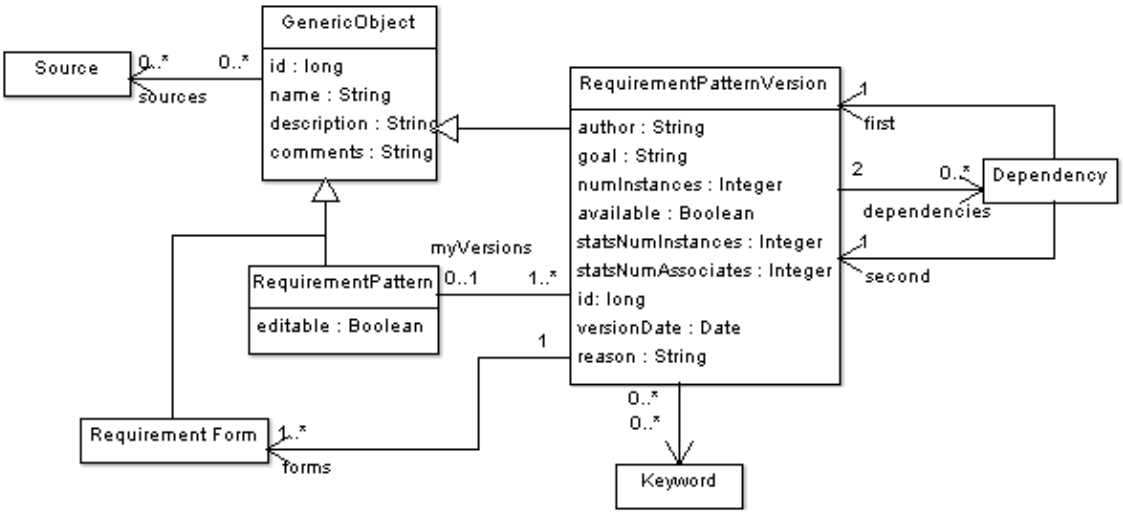


Figure 4.20. RequirementPatternVersion business class and relationships

All these fields are obtained from the RequirementPatternVersion object with the specified {versionid} of the requirement pattern with the specified {patternid}:

- author: References the field author.
- available: References the field available.
- dependencies: Array with a complete representation of the dependencies referenced in the field dependencies. Section 4.6.1.1 (Dependency complete representation).
- forms: Array with a complete representation of the Requirement Forms referenced in the field forms. Section
- goal: References the field goal.
- keywords: Array with the partial representation of the keywords referenced in the field keywords. Section 4.6.1.2 (Keyword partial representation).
- numInstances: References the field numInstances.

- statsNumAssociates: References the field statsNumAssociates.
- statsNumInstances: References the field statsNumInstances.
- versionDate: String that represents the Date referenced in the field versionDate.
- reason: References the field reason.
- requirementPattern: Partial representation of an individual requirement pattern described in section 4.6.2 (Patterns collection).

This field is computed:

- uri: String that references the URI of this individual pattern.

The JSON schema of a complete representation of a Requirement Pattern Version is shown in Figure 4.22.

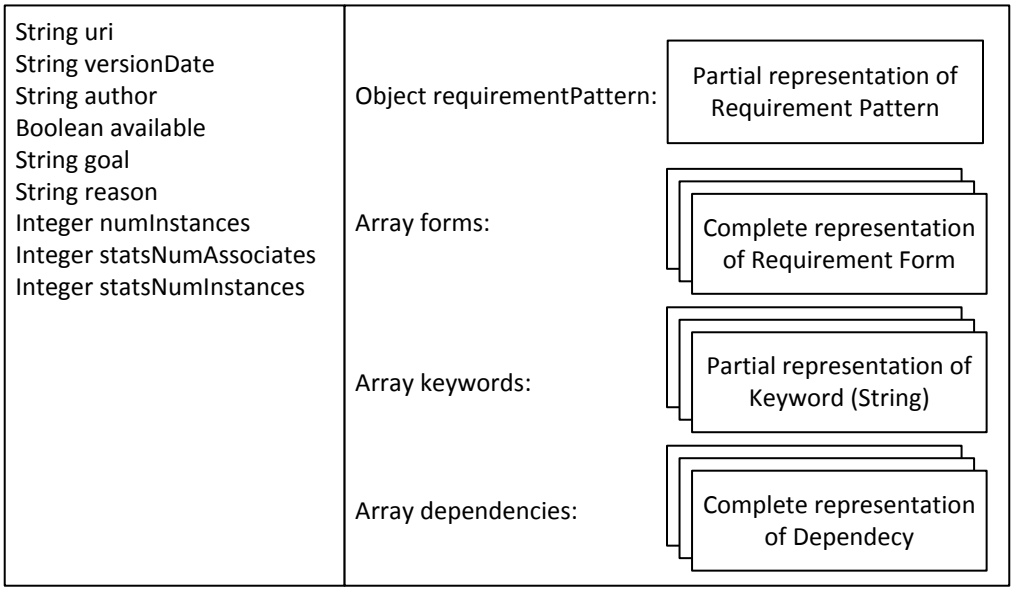


Figure 4.21. Complete representation of Requirement Pattern

4.6.4 Pattern versions collection

This resource is returned when /api/patterns/{patternid}/versions URI is requested.

The pattern versions collection resource provides a list of patterns versions that belong to the specified {patterned] and can or not be filtered using some parameters. It is represented as an array of partial representation of pattern versions.

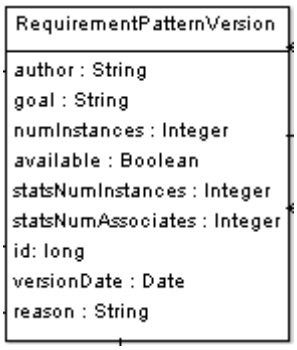


Figure 4.22. RequirementPatternVersion business class.

This partial representation contains the following fields from the RequirementPatternVersion business class (Figure 4.22) objects referenced in "myVersions" field of the RequirementPattern object with the specified {patternid}:

- uri: Computed String that references the URI of this individual pattern version.
- versionDate: References the field versionDate.
- reason: References the field reason.

The JSON schema of a partial representation of a Requirement Pattern Version is shown in Figure 4.23.

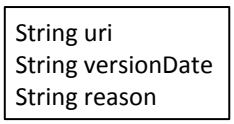


Figure 4.23. Partial representation of Requirement Pattern

4.6.5 Individual Metric

This resource is returned when /api/metrics/{metric} URI is requested.

The individual metric resource provides information about a specific metric and its related metrics and domain values. It is represented as a JSON object that is a complete representations a metric.

Metric business class is an abstract class, so the different concrete classes that can represent a metric are Set, Domain, Integer, String, Float, Time Point and the representation must be different for each concrete class.

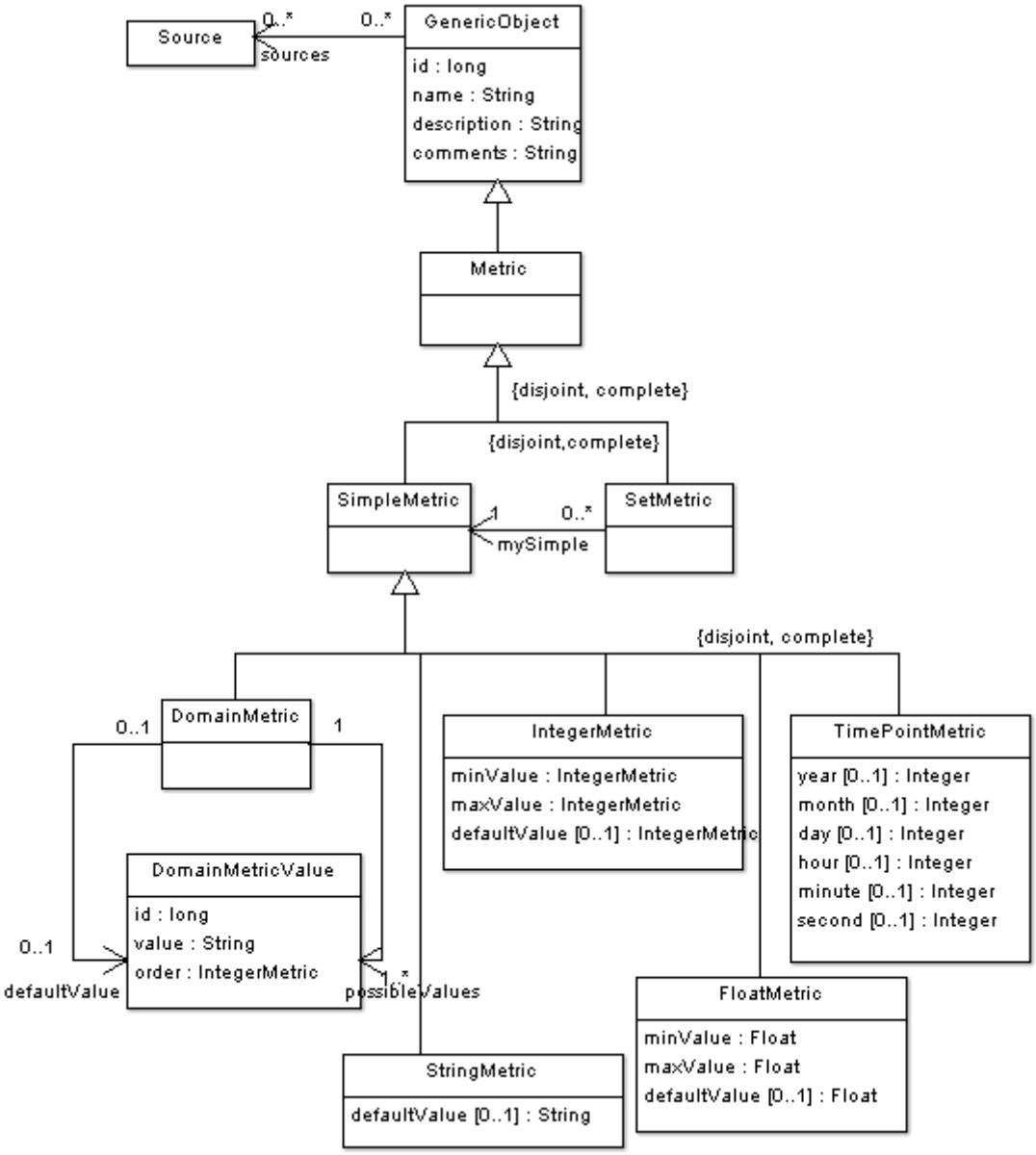


Figure 4.24. Metric business classes' hierarchy

This complete representation of an individual metric is contains a common set of fields based Metric business class interface and a specific set of fields based on the Metric business class hierarchy (Figure 4.24).

Every Metric representation contains the following common set of fields:

- name: References the field name.
- description: References the field description.
- comments: References the field comments.

- sources: Array with a partial representation of the sources referenced in the field "sources" described in section 4.6.8 (Sources collection).
- uri: String that references the URI of this individual metric.

4.6.5.1 Integer Metric complete representation

This complete representation of an integer metric is based in IntegerMetric business class (Figure 4.24) and it will add the following extra fields from the IntergerMetric business class to the generic Metric representation described in section 4.6.5 (Individual Metric):

- minValue: References the field minValue.
- maxValue: References the field maxValue.
- defaultValue: References the field defaultValue.

The JSON schema of a complete representation of an Integer Metric is shown in Figure 4.25.

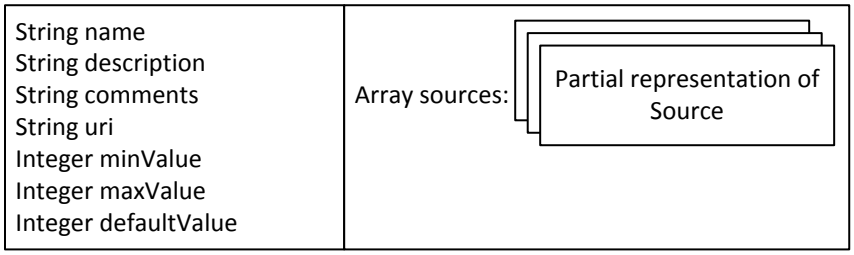


Figure 4.25. Complete representation of Integer Metric

4.6.5.2 Float Metric complete representation

This complete representation of a float metric is based in FloatMetric business class (Figure 4.24) and it will add the following extra fields from the FloatMetric business class to the generic Metric representation described in section 4.6.5 (Individual Metric):

- minValue: References the field minValue.
- maxValue: References the field maxValue.
- defaultValue: References the field defaultValue.

The JSON schema of a complete representation of a Float Metric is shown in Figure 4.26.

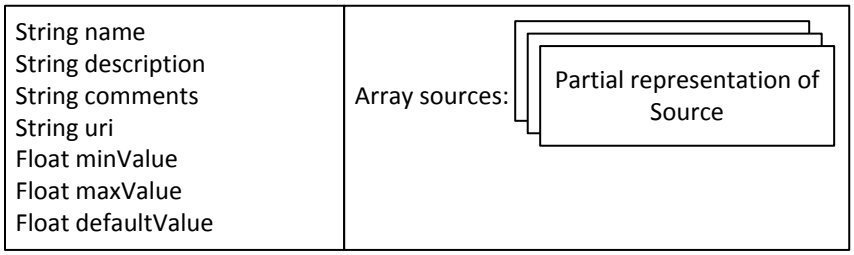


Figure 4.26. Complete representation of Float Metric

4.6.5.3 String Metric complete representation

This complete representation of a string metric is based in StringMetric business class (Figure 4.24) and it will add the following extra fields from the StringMetric business class to the generic Metric representation described in section 4.6.5 (Individual Metric):

- defaultValue: References the field defaultValue.

The JSON schema of a complete representation of a String Metric is shown in Figure 4.27.

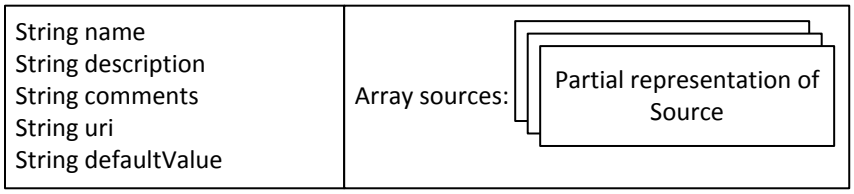


Figure 4.27. Complete representation of String Metric

4.6.5.4 TimePoint Metric complete representation

This complete representation of a time point metric is based in TimePointMetric business class (Figure 4.24) and it will add the following extra fields from the TimePointMetric business class to the generic Metric representation described in section 4.6.5 (Individual Metric):

- year: References the field year.
- month: References the field month.
- day: References the field day.
- hour: References the field hour.
- minute: References the field minute.
- second: References the field second.

The JSON schema of a complete representation of a TimePoint Metric is shown in Figure 4.28.

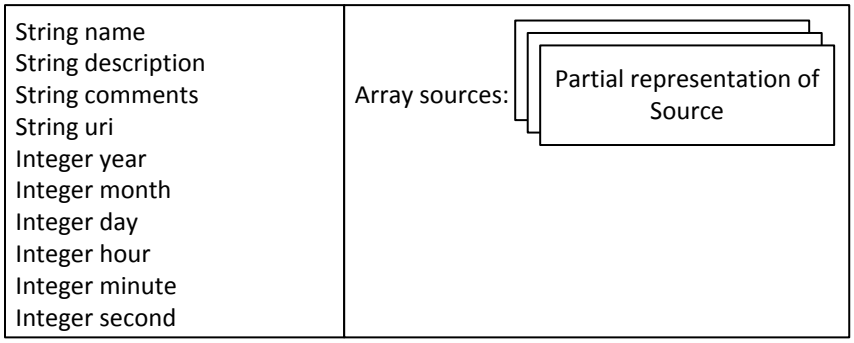


Figure 4.28. Complete representation of TimePoint Metric

4.6.5.5 Domain Metric complete representation

This complete representation of a domain metric is based in DomainMetric business class (Figure 4.24) and it will add the following extra fields from the DomainMetric business class to the generic Metric representation described in section 4.6.5 (Individual Metric):

- defaultValue: References the field value of the DomainMetricValue object referenced in the field defaultValue.
- possibleValues: Array of Strings that reference the field value of the DomainMetricValue objects referenced in the possibleValues relationship ordered using the order field.

The JSON schema of a complete representation of a Domain Metric is shown in Figure 4.29.

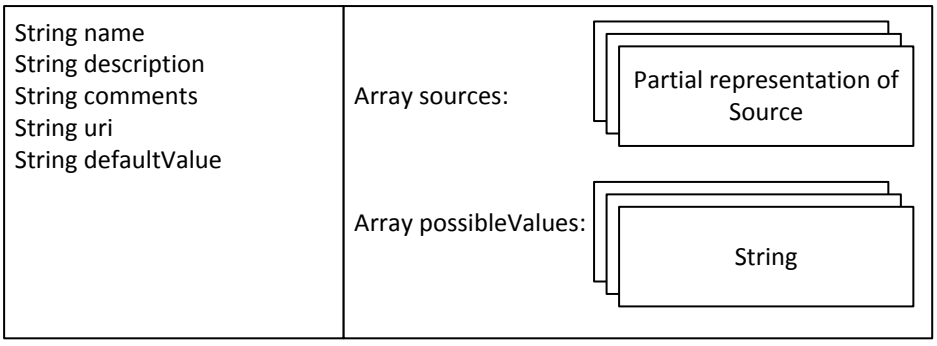


Figure 4.29. Complete representation of Domain Metric

4.6.5.6 Set Metric complete representation

This complete representation of a set metric is based in SetMetric business class (Figure 4.24) and it will add the following extra fields from the SetMetric business class to the generic Metric representation described in section 4.6.3 (Individual Metric):

- simple: A complete representation of the SimpleMetric object referenced in mySimple field. This partial representation is described in the section 4.6.5 (Individual Metric)

The JSON schema of a complete representation of a Set Metric is shown in Figure 4.30.

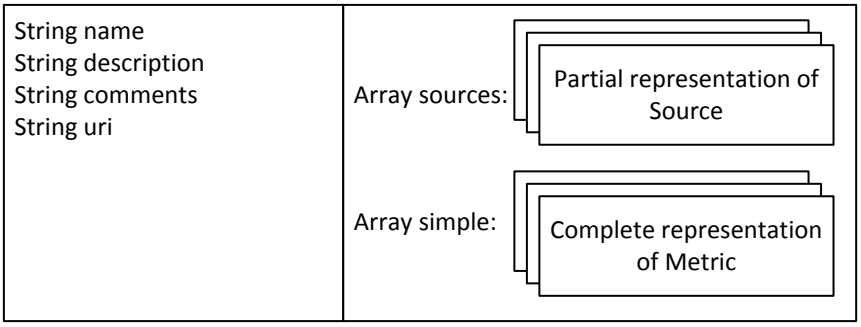


Figure 4.30. Complete representation of Set Metric

4.6.6 Metrics collection

This resource is returned when /api/metrics URI is requested.

The metrics collection resource provides a list of patterns that can or not be filtered using some parameters. It is represented as an array of partial representation of metrics.

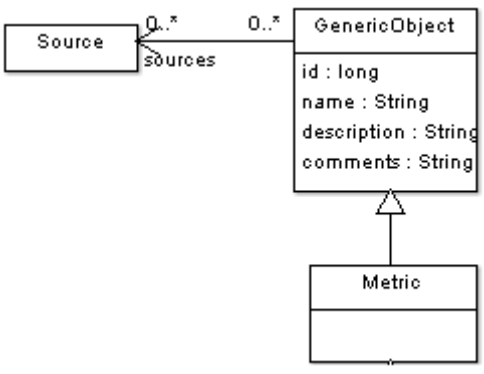


Figure 4.31. Metric business class and relationships

This partial representation of metric is based on the Metric Business class (Figure 4.31) and it contains the fields described below.

- name: References the field name.
- description: References the field description.
- comments: References the field comments.
- sources: Array that contains a partial representation of the sources referenced in the field “sources” described in section 4.6.8 (Sources collection).
- uri: String that references the URI of this individual metric.

The JSON schema of a partial representation of a Metric is shown in Figure 4.32.

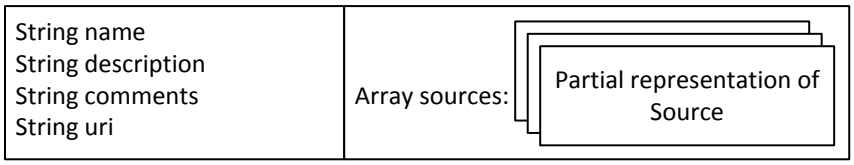


Figure 4.32. Partial representation of Metric

4.6.7 Individual Source

This resource is returned when /api/sources/{sourceid} URI is requested.

The individual source resource provides information about a specific source. It is represented as a JSON object that is a complete representations of a source.

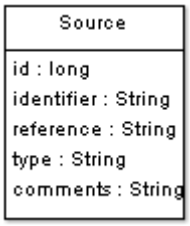


Figure 4.33. Source business class and relationships

This complete representation of an individual source is based on the Source business class (Figure 4.33) and it contains the following fields described below.

- uri: String that references the URI of this individual source.
- identifier: References the field identifier.
- reference: References the field reference.
- type: References the field type.
- comments: References the field comments.

The JSON schema of a complete representation of Source is shown in Figure 4.34.

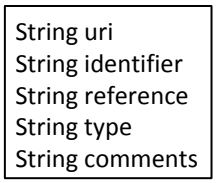


Figure 4.34. Complete representation of Source

4.6.8 Sources collection

This resource is returned when /api/sources URI is requested.

The sources collection resource provides a list of sources that can or not be filtered using some parameters. It is represented as an array of complete representation of sources.

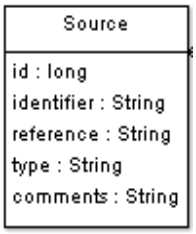


Figure 4.35. Source business class

This partial representation is based on the Source business class (Figure 4.35) and it contains the following fields:

- uri: Computed String that references the URI of this individual source.
- identifier: References the field identifier.

The JSON schema of a partial representation of a Source is shown in Figure 4.36.

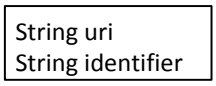


Figure 4.36. Partial representation of Source

4.6.9 Individual Schema

This resource is returned when /api/schemas/{schemaid} URI is requested.

The individual schema resource provides information about a schema, its root classifies, its internal classifiers and its requirement patterns. It is represented as a JSON object that is a complete representations of a schema, its classifiers (roots and internals) and a partial representation of its contained requirement patterns.

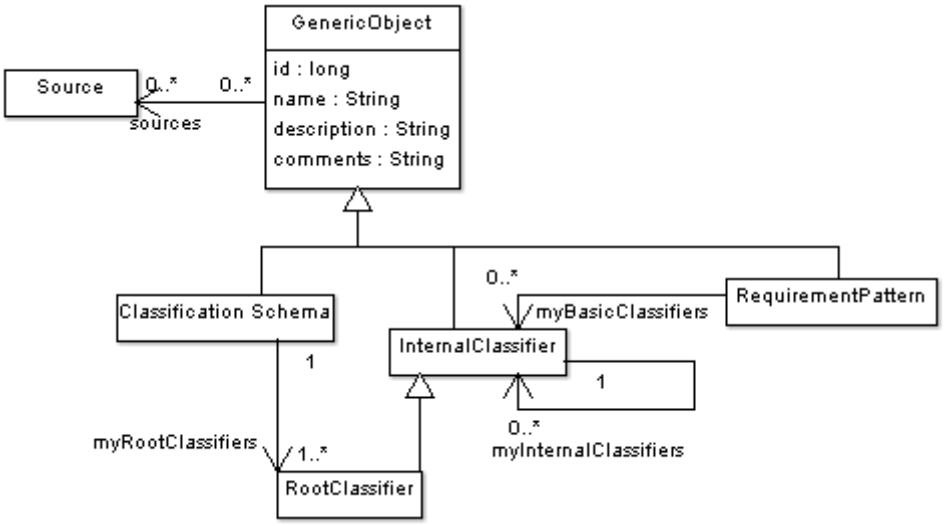


Figure 4.37. ClassificationSchema bussiness class and relationships

This complete representation of a schema is based on the ClassificationSchema business class (Figure 4.37) and it contains the fields described below.

- name: References the field name.
- description: References the field description.
- comments: References the field comments.
- sources: Array that contains a partial representation of the sources referenced in the field “sources” described in section 4.6.8 (Sources collection).
- uri: String that references the URI of this individual schema.
- rootClassifiers: Array that contains a complete representation of the InternalClassifiers referenced in the field “myRootClassifiers” described in section 4.6.11 (Individual Classifier).
- unbindedPatterns: Array that contains a partial representation of the Requirement patterns of the system that are not classified under any internal classifier of this schema described in section 4.6.2 (Patterns collection).

The JSON schema of a complete representation of Schema is shown in Figure 4.38.

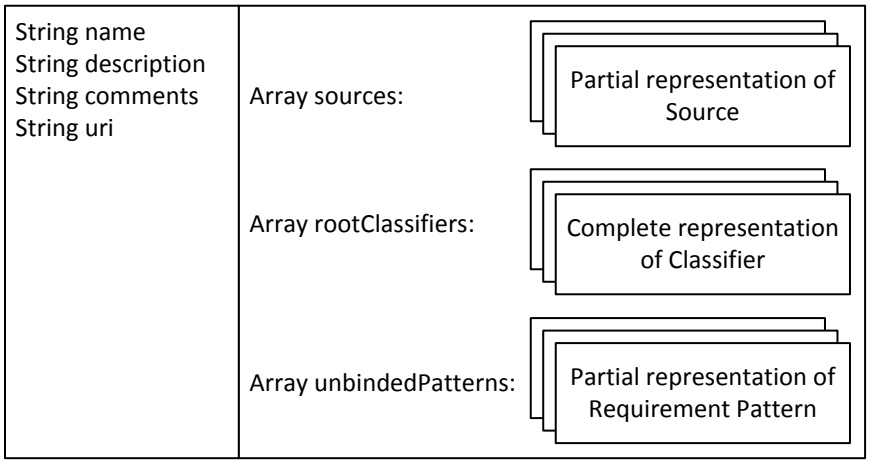


Figure 4.38. Complete representation of Schema

4.6.10 Schemas collection

This resource is returned when /api/schemas URI is requested.

The schemas collection resource provides a list of schemas that can or not be filtered using some parameters. It is represented as an array of partial representation of schemas.

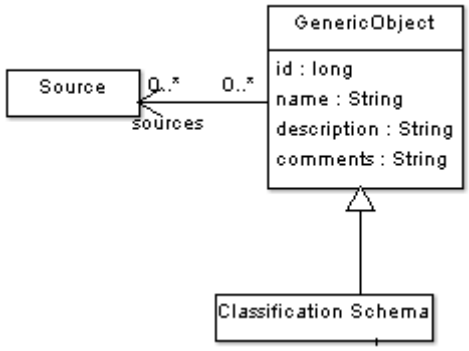


Figure 4.39. ClassificationSchema business class

This partial representation of schemas is based on the Schema Business class (Figure 4.39) and it contains the fields described below.

- name: References the field name.
- description: References the field description.
- comments: References the field comments.
- sources: Array that contains a partial representation of the sources referenced in the field “sources” described in section 4.6.8 (Sources collection).
- uri: String that references the URI of this individual schema.

The JSON schema of a partial representation of Schema is shown in Figure 4.40.

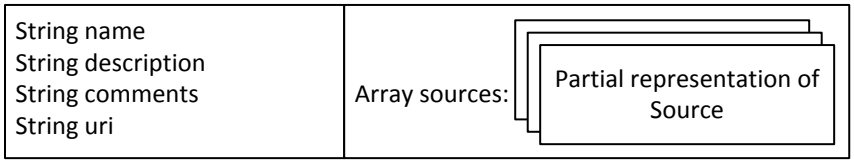


Figure 4.40. Partial representation of Schema

4.6.11 Individual Classifier

This resource is returned when /api/schemas/{schemaid}/classifiers/{classifierid} URI is requested.

The individual classifier resource provides information about a specific classifier, its internal classifiers and its contained requirement patterns. It is represented as a JSON object that is a complete representations of a classifier.

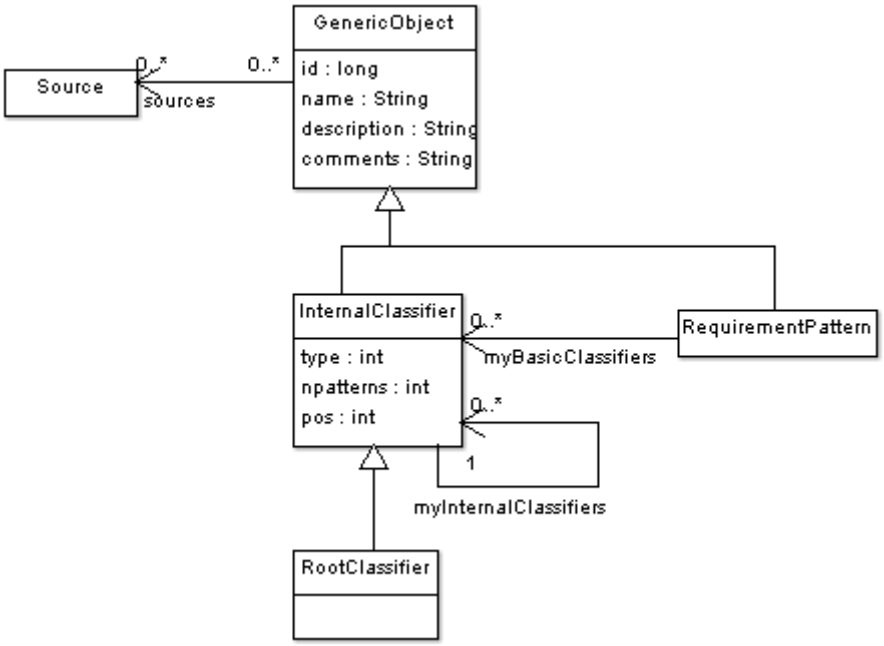


Figure 4.41. InternalClassifier business class and relationships

This complete representation of an individual source is based on the InternalClassifier business class (Figure 4.41) and it contains the following fields described below.

- name: References the field name.
- description: References the field description.
- comments: References the field comments.

- sources: Array that contains a partial representation of the sources referenced in the field “sources” described in section 4.6.8 (Sources collection).
- uri: String that references the URI of this individual schema.
- type: References the field type.
- npatterns: References the field npatterns.
- pos: References the field post.
- internalClassifiers: Array that contains a complete representation of the InternalClassifier objects referred by the field myInternalClassifiers described in section this section (recursive).
- requirementPatterns: Array that contains a partial representation of the RequirementPattern objects that reference the InternalClassifier by their field myBasicClassifiersby the field in described in section (Patterns collection).

The JSON schema of a complete representation of a Classifier is shown in Figure 4.42.

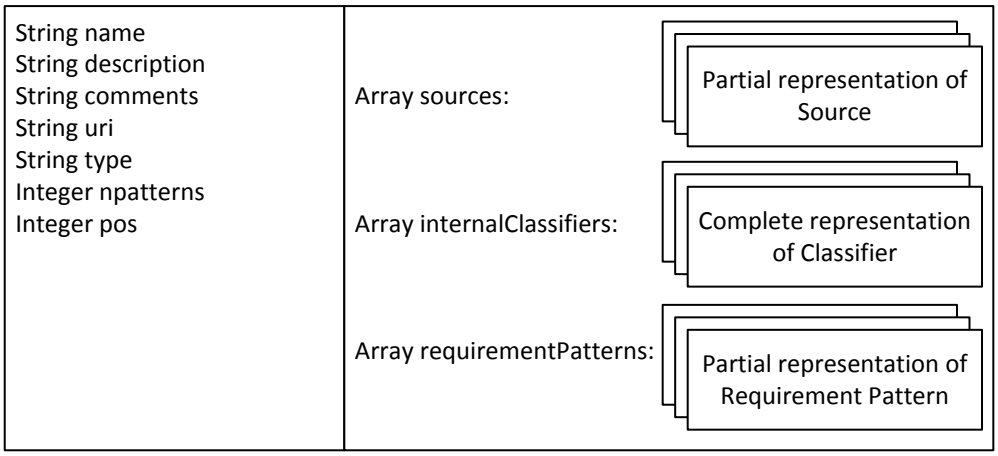


Figure 4.42. Complete representation of Classifier

4.6.12 Classifiers collection

This resource is returned when /api/schemas/{schemaid}/classifiers URI is requested.

The classifiers collection resource provides a list of classifiers contained in the specified {schemaid} that can or not be filtered using some parameters. It is represented as an array of complete representation of classifiers.

This complete representation of classifiers is the same that has been already described in section 4.6.11 (Individual Classifier).

4.7 Resources query parameters

Query parameters are a common type of parameter that are appended to the path of the URL when submitting a request in a query string. The query string is a part of the URL which is passed to the program. Its use permits data to be passed from the HTTP client to the program which generates the web page.

A typical URL containing a query string is as follows:

```
http://server/program/path/?query_string
```

When a server receives a request for such a page, it may run a program, passing the query string unchanged to the program. The question mark is used as a separator and is not part of the query string.

In RESTful services, querying can be performed through the REST API endpoint. REST APIs supports and operates using the core HTTP protocol, and this is the same system used by the client libraries to obtain the view data.

In this project they have been used to filter collections and configure the detail level of the response for some resources. This allow to reduce the size of the response in some cases when the client does not need some information or allow to receive the information dynamically performing new requests to get the individual resources in place of receiving a long complete representation in an unique request.

The resources that allow query string parameters are:

- Metrics collection (/api/metrics):
    - complete={true/false}: The default value if the parameter is not sent is false. This parameter allows to configure the detail level the response. As it has been described in section 4.6.6 (Metrics collection), client receives a partial representation of metrics. If this parameter is set as “true” the response will contain a complete representation of metrics for Set and Domain metrics. This can be useful if the client want to receive a collection that contains information about the relation of sets with other

metrics and the list of values that domain can take in order to show metrics in a tree view.

- Patterns collection (/api/metrics):
    - keyword={string}: The default value if the parameter if not sent is an empty string. This parameter allows to filter the response. As it has been described in section 4.6.2 (Patterns collection), client receives a partial representation of all the patterns of the system ordered alphabetically. If this parameter is set to “true” the response will contain only a collection of requirements patterns that contain the keyword string in their names or their last version keywords.
- Individual schema (/api/schema/{schemaid}):
    - complete={true/false}: The default value if the parameter if not sent is true. This parameter allows to configure the detail level the response. As it has been described in section 4.6.9 (Individual Schema), client receives a complete representation of a schema with the unbinded patterns full tree of classifiers, sub classifiers and the patterns located in the classifiers.

      If this parameter is set to “false” the response will contain only the first hierarchical level of rootClassifiers of the schema. This can be useful if a clients is not interested in the full tree of classifiers and patterns of the schema and only requires information directly related of the schema or is going to load the schema dynamically showing the rest of levels of the tree only when they are requested.

    - unbinded={true/false}: The default value if the parameter if not sent is true. This parameter allows to configure the detail level the response. As it has been described in section 4.6.9 (Individual Schema), client receives a complete representation of a schema with the unbinded patterns full tree of classifiers, sub classifiers and the patterns located in the classifiers.

If this parameter is set to “false” the response will not contain the list unbinded patterns and “unbindedPatterns” field will not be returned in the response. This can be useful if a clients is not interested in the list of unbinded patterns or he is going to load it later, only when is specifically requested.

- Classifiers collection (/api/schema/{schemaid}/classifiers):
    - complete={true/false}: The default value if the parameter if not sent is true. This parameter allows to configure the detail level the response. As it has been described in section 4.6.12 (Classifiers collection), client receives an array of complete representations of schema classifiers with a full tree of sub classifiers and the patterns contained in the classifier.

      If this parameter is set to “false” the response will contain only the first hierarchical level of root classifiers of the schema. This can be useful in the same situations that using “complete=false“ in Individual Schema resource.

- Individual Classifier (/api/schema/{schemaid}/classifier/{classifierid}):
    - complete={true/false}: The default value if the parameter if not sent is true. This parameter allows to configure the detail level the response. As it has been described in section 4.6.11 (Individual Classifier), client receives a complete representation of a classifier with a full tree of sub classifiers and the patterns contained in the classifier.

      If this parameter is set to “false” the response will contain only the first hierarchical level of sub classifiers contained in the specific classifier. This can be useful in the same situations that using “complete=false” in Individual Schema resource.

5 Development of Pabre-WS

Once decisions about the requirements that the Pabre-WS must satisfy, what technologies to use in the development and also about the organization, representation of resources and parameters that will guide the implementation of the web service have been established, this section presents and describes the implementation decisions and the process that has been followed during the development process of the web services.

5.1 Environment preparation

The first step in the development was to prepare the development environment. The creation the new project in the context of the Eclipse IDE has two requirements:

- It must be compatible with Apache Maven.
- It must be a J2EE Web Application.

In order to reach these goals the m2eclipse and m2e-wtp plugins (as explained in section 4.3.7 Apache Maven) need to be installed in Eclipse to support the creation of a Web Application Maven project. Then, a new Maven Project can be created in Eclipse and then the “maven-archetype-webapp” must be selected in the archetype selection step.

Once, the project is created the Projects Facets applied in eclipse have been the followings (they can be selected in the Proprieties of the project at the Project Facets section):

- Dynamic Web Module. Version 3.0
- Java. Version 1.6
- JavaScript. Version 1.0
- JAX-RS (Rest Web Services). Version 1.1 (The “Update deployment descriptor” will be deactivated because it will updated manually)

Since all the dependencies are going to be provided by maven, only Server runtime libraries needs to be added to project classpath. To configure it, it is necessary to go to Project properties:

- In the “Java Build Path” section it will used the “Add Library…” button and select to add “Server Runtime” selecting some server runtime, in this case is Tomcat, to add their runtime libraries to the classpath of the project as shown in Figure 5.1.

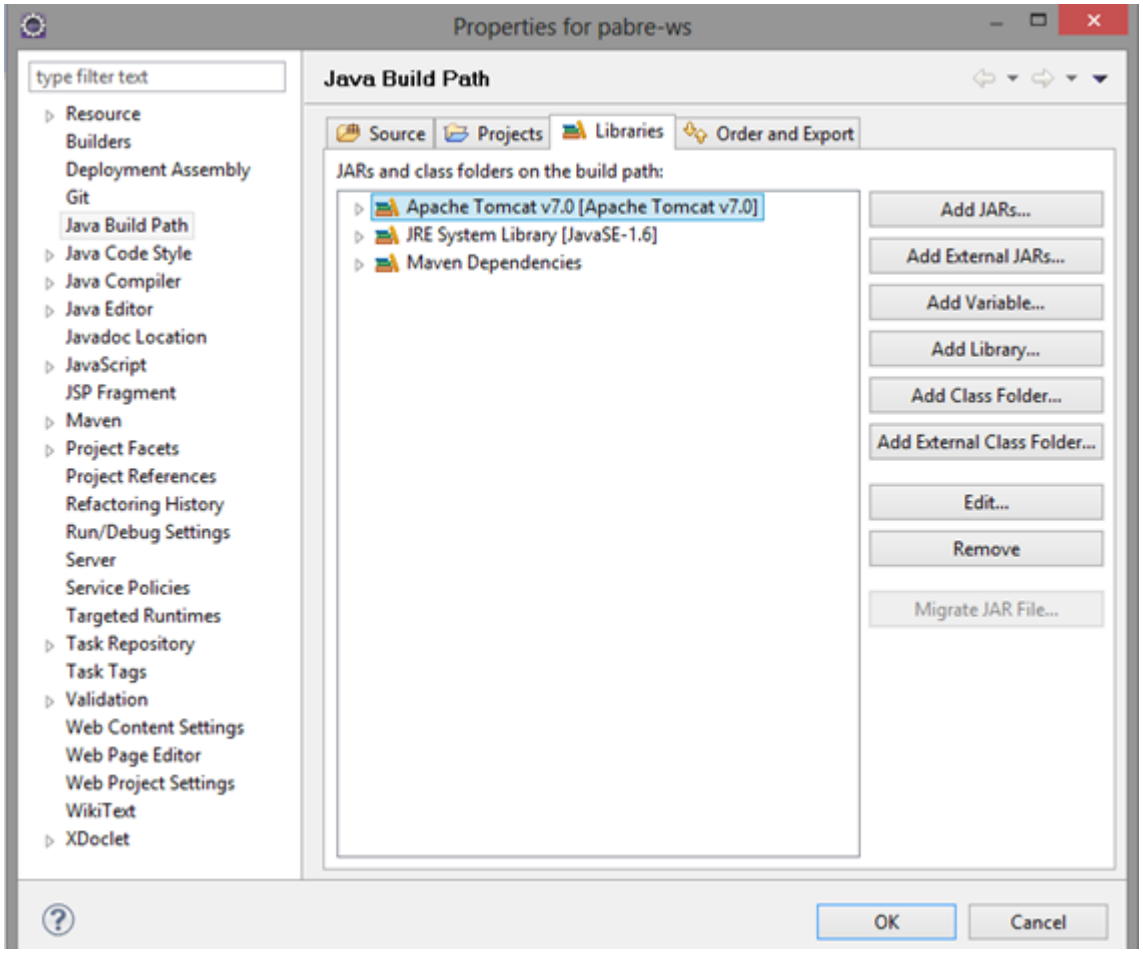


Figure 5.1. “Web App” and “Server Runtime” libraries added to classpath

After these steps the Eclipse project is configured to support correctly the Maven standard layout in a Java2EE project.

5.2 Apache Maven configuration

To configure Apache Maven in the context of this project it is necessary to write the pom.xml configuration file stored in the root folder of the project.

The pom.xml file with the explanation of each configuration option selected in the file is detailed below these lines:

- This is the header of the pom.xml. In this section variables such as the name of the project, the version, the type of packaging, etc. is configured

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>Pabre-WS</groupId>
  <artifactId>Pabre-WS</artifactId>
  <packaging>war</packaging>
  <version>1.0</version>
  <name>Pabre-WS Maven Webapp</name>
```

- The properties section allows developers to store some variables. This is very useful in order to easily change the version of some dependencies we are going to use in our project. In this case, we are going to use jersey version 1.17. If in the future we would want to change to a different version, by changing the value of this properties, maven would automatically solve the dependencies of the new version, download and upgrade them in the project. Also project.buid.sourceEncoding and project.resources.sourceEnconding proprieties must be configured to set the character encoding of the source and resource files to UTF-8 in order to avoid platform problems in different systems.

```
  <properties>
      <jersey-version>1.17</jersey-version>
      <project.build.sourceEncoding>
      UTF-8
      </project.build.sourceEncoding>
      <project.resources.sourceEncoding>
      UTF-8
      </project.resources.sourceEncoding>
  </properties>
```

- The dependencies section of the pom.xml is where are the dependencies of the project are declared. In this section the path and the name and the version of each dependency is specified. As it can be observed, in jersey dependencies the version is specified through the propertied previously declared in the previous section.

  The dependencies declared in this section are:

- jersey-server: Is the library used to implement the rest layer in the web service server side.
- jersey-servlet: Is the library needed to deploy an application on a servlet container.
- jersey-server-linking: This library is needed to have declarative hyperlinking support in jersey. This feature is described in section 5.6.1.1 (Declarative Hyperlinking annotations).
- jackson-jaxrs-json-provider: This dependency is needed to update the Jackson version that comes with jersey 1.17 and belongs to the 1.x branch to Jackson 2.1.4 that belongs to the 2.x branch. This new version is major update that comes with some set of features that are going to be used in this project as described in section 4.3.5 (Jackson JSON processor)
- mysql-connector-java: This dependency is needed to have the MySQL JDBC connector driver to connect to MySQL databases.
- hibernate-core: Needed to have hibernate support on data layer for code reused from previous PABRE tools.
- derbyclient: Is the Apache Derby JDBC connector driver to be able to connect Apache Derby DBSM, it has been declared to have also support with this database simply switching the Hibernate configuration file.
- c3p0: Is the library needed to implement the c3p0 connection pooling algorithm described in section 6.3.4 (Hibernate JDBC connection pool).
- hibernate-c3p0: This library is needed to connect Hibenate with the c3p0 connection pooling algorithm.
- servlet-api: This library, needed to implement the ServletContextListener that initializes and frees the application resources on application load and unload, should be provided by the servlet container application (Tomcat or others), but it has to be declared in the dependencies section configuring its “scope” property to “provided” in order to make maven know its availability during the build process.
- slf4j-log4j: Is the slf4j to log4j bridge library needed to use the log4j implementation over slf4j as a logging strategy, required by hibernate and others libraries.

- javassist: is a Java library providing a means to manipulate the Java bytecode of an application and support for structural reflection (ability to change the implementation of a class at run time) required by Hibernate.

```
<dependencies>
 <dependency>
     <groupId>com.sun.jersey</groupId>
     <artifactId>jersey-server</artifactId>
     <version>${jersey-version}</version>
 </dependency>
 <dependency>
     <groupId>com.sun.jersey</groupId>
     <artifactId>jersey-servlet</artifactId>
     <version>${jersey-version}</version>
 </dependency>
 <dependency>
       <groupId>com.sun.jersey</groupId>
       <artifactId>jersey-server-linking</artifactId>
       <version>${jersey-version}</version>
 </dependency>
 <dependency>
       <groupId>com.fasterxml.jackson.jaxrs</groupId>
       <artifactId>jackson-jaxrs-json-provider</artifactId>
       <version>2.1.4</version>
 </dependency>
 <dependency>
       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
       <version>5.1.14</version>
 </dependency>
 <dependency>
       <groupId>org.hibernate</groupId>
       <artifactId>hibernate-core</artifactId>
       <version>3.6.10.Final</version>
 </dependency>
 <dependency>
       <groupId>org.apache.derby</groupId>
       <artifactId>derbyclient</artifactId>
       <version>10.10.1.1</version>
 </dependency>
 <dependency>
       <groupId>com.mchange</groupId>
       <artifactId>c3p0</artifactId>
       <version>0.9.2.1</version>
 </dependency>
 <dependency>
       <groupId>javax.servlet</groupId>
       <artifactId>servlet-api</artifactId>
       <version>2.5</version>
       <scope>provided</scope>
 </dependency>
 <dependency>
       <groupId>org.slf4j</groupId>
       <artifactId>slf4j-log4j12</artifactId>
       <version>1.6.1</version>
 </dependency>
 <dependency>
       <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-c3p0</artifactId>
        <version>3.6.10.Final</version>
    </dependency>
    <dependency>
        <groupId>org.javassist</groupId>
        <artifactId>javassist</artifactId>
        <version>3.16.1-GA</version>
    </dependency>
  </dependencies>
```

- This section is related to the compilation and build phase of the project. Here is declared the name of the bundled project when it is finally built and the used plugins. In this case the maven-compiler-plugin is used in order to indicate Maven that the source code and the compilation must be made for Java 1.6 version. In other case, the default behavior of Maven is to perform it for Java 1.5 version and this makes some compatibility problems with the Eclipse configuration.

```
  <build>
    <finalName>Pabre-WS</finalName>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
          <configuration>
            <source>1.6</source>
            <target>1.6</target>
          </configuration>
        </plugin>
      </plugins>
  </build>
</project>
```

5.3 Reused code from Pabre-Man

In order to save time and developing time, it was taken the decision of reusing some Java classes from Pabre-Man. After an analysis of Pabre-Man classes it was decided to reuse the classes of the domain and data layer of that software since the business and data logic is going to remain preserved in the web service.

Thanks to the tree-tier architecture it was very easy to isolate them because they were classified in a set of Java packages. After discarding the set of packages related with gui,

reports and statistics, it was decided to make use of the following Java packages copying them in the application sources folder (src/main/java) of the project

- edu.upc.gessi.rptool.data
- edu.upc.gessi.rptool.data.controller
- edu.upc.gessi.rptool.domain
- edu.upc.gessi.rptool.domain.metrics
- edu.upc.gessi.rptool.domain.requirementPatterns
- edu.upc.gessi.rptool.domain.schema

In addition to the set of classes some libraries used in Pabre-Man also are needed since they are used from reused classes. The required libraries are:

- antlr-2.7.6.jar
- commons-beanutils.jar
- commons-collections-3.1.jar
- commons-digester-1.7.jar
- commons-logging.jar
- derby.jar
- derbyclient.jar
- derbyLocale_es.jar
- derbynet.jar
- dom4j-1.6.1.jar
- hibernate3.jar
- javassist-3.4.GA.jar
- jta-1.1.jar
- log4j-1.2.15.jar
- slf4j-api-1.5.2.jar
- slf4j-log4j12-1.5.2.jar

But, in place of copying the libraries' jars to the new project, they have been configured in the Maven dependencies section in order to allow Maven to download and manage

them, declaring all the required dependencies as described in section 5.2 (Apache Maven configuration), resulting in a reduced set of dependencies as shown in Figure 5.2.

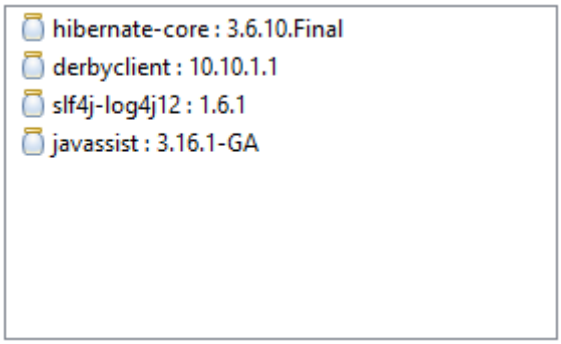


Figure 5.2. Maven configured POM dependencies.

As it can be observed in Figure 5.2 the dependencies version have been upgraded to newer versions taking profit of this process, since it has been tested that they are fully compatible with previous code.

- Hibernate has been upgraded from version 3.3.2 to version 3.6.10.
- Derby client JDBC connector driver has been upgraded from 10.4.2.0 to 10.10.1.1.
- Slf4j-log4j12 bridge and log4 have been upgraded from 1.5.2 to 1.6.1.
- Javassist has been upgraded from 3.4 to 3.16.1.

Finally, configuration files must be copied into the new project. These files are stored in the "/config" folder of Pabre-Man. Since, these files are considered resources, this folder must be copied in the "/src/main/resources" location, having all the configuration files of the new project in "/src/main/resources/config". Also, hibernate mapping file, that was stored in has been moved to this folder from "/src/main/java/edu/upc/gessi/rptool/domain/Mapeig.hbm.xml" since Maven documentation explains that only java source files should be stored in "/src/main/java" folder and this file is considered a resource.

The folder content after this process is:

- hibernate.cfg.xml
- log4j.cfg.xml

- schema.cfg.xml
- Mapeig.hbm.xml

Once all these files have been copied in the new project we will have all the business logic and the data layer prepared to work with the Derby database of Pabre-Man.

5.4 Jersey configuration

In this section the preparation and the configuration of Jersey will be described. The goal of using Jersey is to simplify development of RESTful Web services solving all the tasks related with the request and response management.

Jersey implements support for the annotations defined in JSR-311, making it easy for developers to build RESTful web services with Java. The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services.

So, in place of having to write servlets that manage the HTTP requests and responses and find the correct classes that process the requests, developers decorate Java programming language class files with HTTP-specific annotations to define resources and the actions that can be performed on those resources.

Figure 5.3 (Jersey architecture) describes the architecture that will be used.

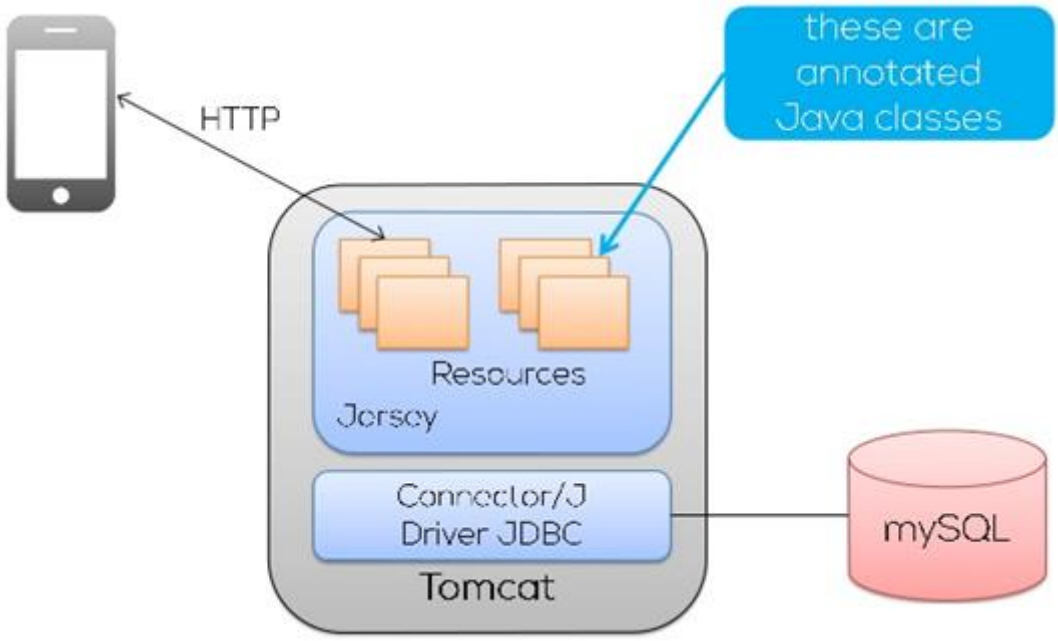


Figure 5.3. Jersey architecture

Jersey implements its own servlet that will located in an Apache Tomcat servlet container. This servlet can be configured to track some specific classes or packages to look for the annotated resource classes. When a request arrives to the servlet, it will use the URI to look for the correct resource that will receive the request, will process it making use of the business logic layer and will build the response that will be send to the client.

The first step to make all this work in the project (once all the dependencies have been solved using maven as explained in the previous section) is to edit the "/src/main/webapp/WEB-INF/web.xml" file to configure the servlet mapping and the resource classes.

Below this line, the "web.xml" file configuration is described and explained:

- This section is the header of the web.xml file, here is declared the Web application display name

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
      version="2.5">
   <display-name>Pabre-WS</display-name>
```

- This section is where the Jersey servlet is declared. Basically, the Jersey servlet container class is declared as a servlet and some parameters of the class are configured. In this case two parameters have been configured:
    - com.sun.jersey.config.property.packages: The value is configured to the name of the package where the annotated resource classes are located.

      In Pabre-WS, all the resource classes are locate in the "edu.upc.gessi.rptool.rest" package. This is the package that will be tracked by Jersey servlet when the application boots in order to decide what class is going to process the requests based on the URI.

    - com.sun.jersey.spi.container.ContainerResponseFilters: This parameter allow to configure what Container filters are going to be used, which are filters that are registered to filter the response after the response has returned from a resource method.

      In Pabre-WS "com.sun.jersey.server.linking.LinkFilter" filter is going to be used to allow Declarative Hyperlinking.

  Finally the option "load-on-startup" is set to 1 in order to assure that the servlet is always going to initialize before any other servlet.

```
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletCon
tainer</servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param
-name>
            <param-value>edu.upc.gessi.rptool.rest</param-value>
        </init-param>
        <init-param>
            <param-name>com.sun.jersey.spi.container.ContainerResponse
Filters</param-name>
            <param-value>com.sun.jersey.server.linking.LinkFilter</par
am-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

- In this section the servlets of the application are mapped to a specific URI. In this case, all the resource URIs are going to be located under the /api path of the servlet.

  This will allow in the future a legacy support if new versions of the api are deployed. In this case, a new jersey servlet can be configured to track another different package with the new version resource classes and can be mapped to a new url-pattern such as "/apiv2/*".

```
    <servlet-mapping>
      <servlet-name>Jersey  REST  Service</servlet-name>
      <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
```

- Finally, a welcome file list is specified. When the root path of the application is requested the server will look for the files in this ordered list and the request will be redirected to the first available URI in this list.

  Since, this is a web service, no welcome file is needed. But this welcome file list can be configured in order to redirect the request to the web service client that is going to be developed to test the web service.

```
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

5.5 Resource classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with JAX-RS annotations. A root resource class is must be annotated with @Path, have at least one method annotated with @Path or a resource method designator annotation such

as @GET, @PUT, @POST, @DELETE. Resource methods are methods of a resource class annotated with a resource method designator.

As it has been previously explained, all the root resource classes are located in the "edu.upc.gessi.rptool.rest" package. This package contains 4 annotated root resource classes:

- edu.upc.gessi.rptool.rest.Metrics
- edu.upc.gessi.rptool.rest.Patterns
- edu.upc.gessi.rptool.rest.Schemas
- edu.upc.gessi.rptool.rest.Sources

Each one of these classes has been designed following the addressing schema described in section 4.5 (Resources addressing schema) and they have one method for each of the possible resources that can be requested.

5.5.1 Used annotations

The annotations that have been used in these classes are the following (JerseyTeam, s.f.):

- Relative uri annotation (@Path)

  The @Path annotation's value is a relative URI path. When a class is annotated and the URI requested by the client matches with the specified @Path that resource class will be found by the Jersey servlet and the correct resource method depending on the HTTP request method will be executed. For example:

```
@Path("/patterns")
```

  A very useful point of JAX-RS is that embedded variables can be used in the URIs. URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example:

```
@Path("/patterns/{patternid}")
```

Regular expressions also can be used in URI path templates to restrict the set of possible values of the embedded variables, but they have not been used in the context of this project.

A @Path value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path value may or may not end in a '/', it makes no difference, and thus request URLs that end or do not end in a '/' will both be matched.

@Path may also be used on methods of root resource classes. This enables common functionality for a number of resources to be grouped together and potentially reused. When @Path may be used is on resource methods those methods are referred to as sub-resource methods.

- Resource method designator (@GET)

  @GET, @PUT, @POST, @DELETE and @HEAD are resource method designator annotations defined by JAX-RS. They correspond to the HTTP Methods with the same name. The selected resource method of a resource class, and then, the behaviour of a resource is determined by which HTTP methods the resource class is responding to.

  In the context of this project only @GET annotation has been used since only query operations are allowed by clients, so every resource method is annotated as:

```
@GET
```

- @Produces

  The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. @Produces can be applied at both the class and method levels. If a resource class is capable of producing more than one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared

by the client. More specifically the Accept header of the HTTP request declares what is most acceptable.

Since in this project, only JSON format is considered as the MIME media type for representing the resources, every resource method is annotated as:

```
@Produces({MediaType.APPLICATION_JSON})
```

- Parameters annotations (@PathParam,@DefaultValue and @QueryParam)

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request.

@PathParam is used to extract a path parameter from the path component of the request URL that matched the path declared in @Path. For example, for the declared path:

```
@Path("/patterns/{patternid}")
```

The @PathParam annotation used to store the parameter in a specific variable would be:

```
public Pattern getPattern(@PathParam("id") long id)
```

On the other hand, @QueryParam is used to extract query parameters from the Query component of the request URL and if the query parameter does not exist then the default value declared in the @DefaultValue annotation, will be assigned to the variable. For example:

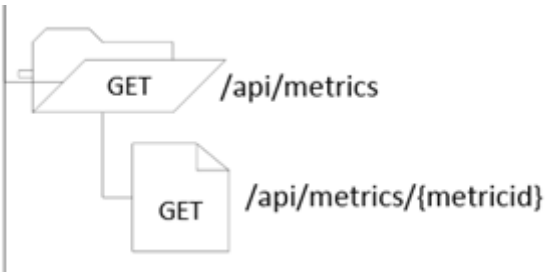
```
public SchemaDTO getSchema(
      @DefaultValue("true") @QueryParam("complete")
      boolean complete
)
```

5.5.2 Metrics root resource class

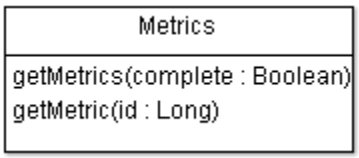
As it has been described in section 4.5 (Resources addressing schema), metrics resources were designed to support two different URIs:

- /api/metrics to get a collection of metrics.

- /api/metrics/{metricid} to get a specific metric by its id.



The resulting root resource class implementation is edu.upc.gessi.rptool.rest.Metrics with one resource method for each URI:



- getMetrics() method is related to /api/metrics URI and returns a collection of metrics
- getMetrics() method is related to /api/metrics/{metricid} URI and returns an individual metric with the specified id.

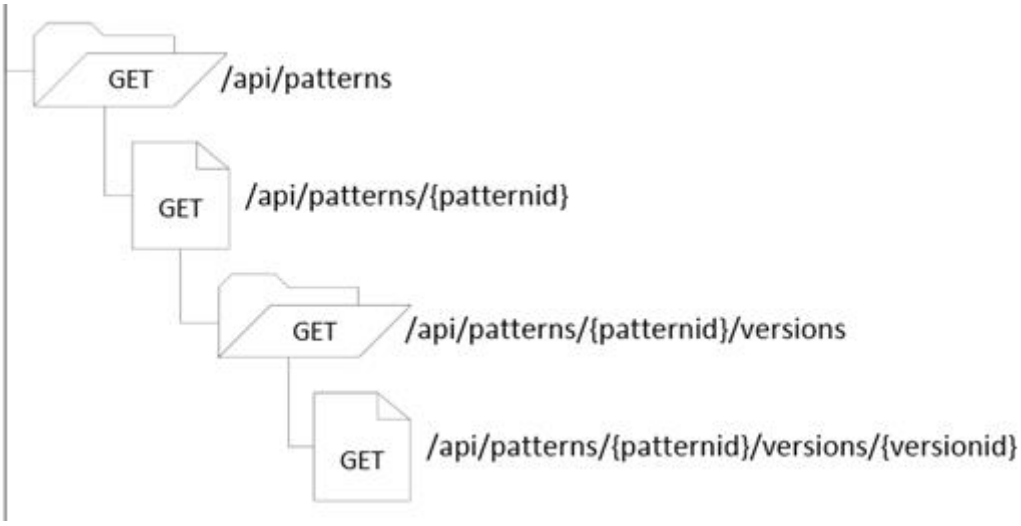
In the signature of the class it can be observed the annotations described in section 5.5.1 (Used annotations) that it have been used:

```
@Path("/metrics")
public class Metrics {

  @GET
  @Produces({MediaType.APPLICATION_JSON})
  public List<MetricDTO> getMetrics(
    @DefaultValue("false") @QueryParam("complete") boolean complete
    )

  @GET
  @Path("{id}")
  @Produces({MediaType.APPLICATION_JSON})
  public MetricDTO getMetric(@PathParam("id") long id)

}
```

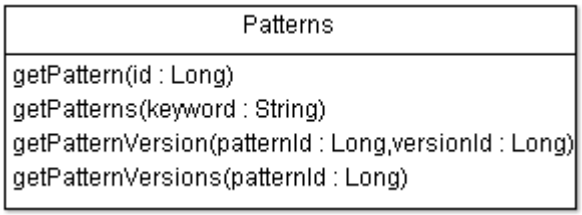
5.5.3 Patterns root resource class

As it has been described in section 4.5 (Resources addressing schema), patterns resources were designed to support four different URIs:

- /api/patterns to get a collection of patterns.
- /api/patterns/{patternid} to get an individual pattern by its id.
- /api/patterns/{patternid}/versions to get a collection of patter versions by the id of their pattern.
- /api/patterns/{patternid}/versions/{versioned} to get an individual pattern version by its id and the id of the pattern it belong to.



The resulting root resource class implementation is edu.upc.gessi.rptool.rest.Patterns with one resource method for each URI:



In the signature of the class it can be observed the annotations described in section 5.5.1 (Used annotations) that it have been used:

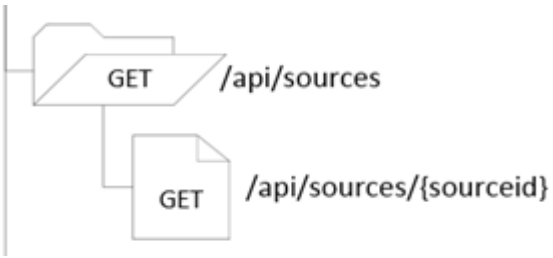
```
@Path("/patterns")
public class Patterns {

  @GET
  @Produces({MediaType.APPLICATION_JSON})
  public Set<RequirementPatternDTO> getPatterns(
      @DefaultValue("") @QueryParam("keyword") String keyword
  )

  @GET
```

```
  @Path("{id}")
  @Produces({MediaType.APPLICATION_JSON})
  public RequirementPatternDTO getPattern(@PathParam("id") long id)

  @GET
  @Path("{patternId}/versions")
  @Produces({MediaType.APPLICATION_JSON})
  public Set<VersionDTO> getPatternVersions(
      @PathParam("patternId") long patternId
  )

  @GET
  @Path("{patternId}/versions/{versionId}")
  @Produces({MediaType.APPLICATION_JSON})
  public RequirementPatternVersionDTO getPatternVersion(
      @PathParam("patternId") long patternId,
      @PathParam("versionId") long versionId
  )

}
```

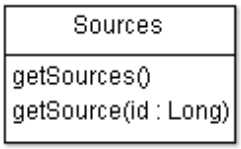
5.5.4 Sources root resource class

As it has been described in section 4.5 (Resources addressing schema), sources resources were designed to support two different URIs:

- /api/sources to get a collection of sources.
- /api/sources/{sourceid} to get an individual source by its id.



The resulting root resource class implementation is edu.upc.gessi.rptool.rest.Sources with one resource method for each URI:



In the signature of the class it can be observed the annotations described in section 5.5.1 (Used annotations) that it have been used:

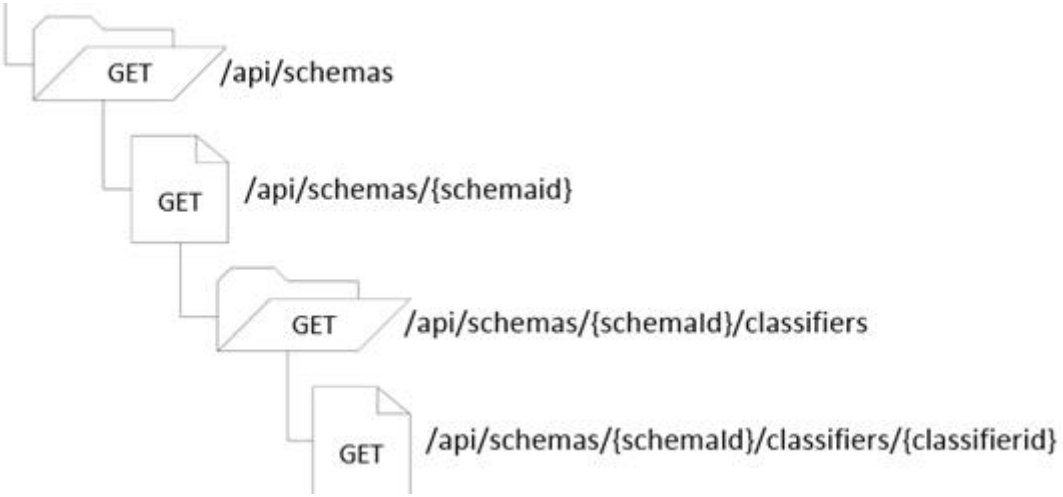
```
@Path("/sources")
```

```
public class Sources {

  @GET
  @Produces({MediaType.APPLICATION_JSON})
  public List<SourceDTO> getSources()

  @GET
  @Path("{id}")
  @Produces({MediaType.APPLICATION_JSON})
  public SourceDTO getSource(@PathParam("id") long id)

}
```

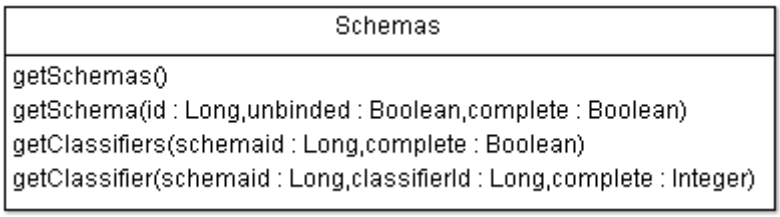
5.5.5 Schemas root resource class

As it has been described in section 4.5 (Resources addressing schema), schemas resources were designed to support four different URIs:

- /api/schemas to get a collection of schemas.
- /api/schemas/{schemaid} to get an individual schema by its id.
- /api/schemas/{schemaid}/classifiers to get a collection of classifiers by the id of their schema.
- /api/schemas/{schemaid}/classifiers/{classifierid} to get an individual classifier by its id and the id of the schema it belongs to.



The resulting root resource class implementation is edu.upc.gessi.rptool.rest.Sources with one resource method for each URI:



In the signature of the class it can be observed the annotations described in section 5.5.1 (Used annotations) that it have been used:

```
@Path("/schemas")
public class Schemas {

  @GET
  @Produces({MediaType.APPLICATION_JSON})
  public List<SchemaDTO> getSchemas()

  @GET
  @Path("{id}")
  @Produces({MediaType.APPLICATION_JSON})
  public SchemaDTO getSchema(
       @PathParam("id") long id,
       @DefaultValue("true") @QueryParam("unbinded") boolean unbinded,
       @DefaultValue("true") @QueryParam("complete") boolean complete
  )

  @GET
  @Path("{schemaid}/classifiers")
  @Produces({MediaType.APPLICATION_JSON})
  public Set<RootClassifierDTO> getClassifiers(
       @PathParam("schemaid") long schemaid,
       @DefaultValue("true") @QueryParam("complete") boolean complete
  )

  @GET
  @Path("{schemaid}/classifiers/{classifierid}")
  @Produces({MediaType.APPLICATION_JSON})
  public InternalClassifierDTO getClassifier(
       @PathParam("schemaid") long schemaid,
       @PathParam("classifierid") long classifierid,
       @DefaultValue("true") @QueryParam("complete") boolean complete
  )

}
```

5.6 DTO classes (Jackson JSON processing)

In a RESTful Web Service, clients are going to work with a remote interface, such as Remote Façade, in this context, each call to it is expensive. As a result a RESTful API needs to reduce the number of calls, and that means that it needs to transfer more data with each call. The solution is to create a Data Transfer Object that can hold all the data

for the call. It needs to be serializable to go across the connection. Although the main reason for using a Data Transfer Object is to batch up what would be multiple remote calls into a single call, the most important motivation is the advantage they provide to encapsulate the serialization mechanism for transferring data over the network. By encapsulating the serialization like this, the DTOs keep this logic out of the rest of the code and also provide a clear point to change easily the serialization structure when is needed. (Fowler, 2002)

As it has been previously explained in section 4.2.2 (JSON format) and 4.3.5 (Jackson JSON processor) the serialization mechanism that is used in this project is a JSON processor that automatically maps objects to JSON format before send them to the client.

In order to adapt objects from the business layer to the desired JSON structure in the responses of the web service, a hierarchy of DTO classes has been defined in the packages edu.upc.gessi.rptool.dtos and edu.upc.gessi.rptool.dtos.metrics.

This hierarchy (Figure 5.4) allows to easily create the objects that are going to be mapped to the final JSONs that are going to be returned to clients.

The general structure of these DTO classes is set of attributes and relationships with other DTO classes and some constructors:

- A constructor that receives as a parameter a business object in order to retrieve the needed information from it and initializes the necessary attributes and related DTOs of itself.
- A constructor that receive a set of parameters with the values of the attributes and/or the related DTOs and initializes the necessary attributes and related DTOs of itself.

Once the construction of the DTO and its related DTOs is finished, the object can be automatically mapped into a JSON using the Jackson JSON processor.

5.6.1 Used annotations

The annotation that have been used in these classes are (Jackson team, 2013):

- @JsonInclude: annotation used to define if certain "non-values" (nulls or empty values) should not be included when serializing; can be used on per-property basis as well as default for a class (to be used for all properties of a class).

  For example, using @JsonInclude(Include.NON_NULL) an attributed with a null value would not be include in the serializes JSON.

  The possible values for this annotation are:

  - ALWAYS: Value that indicates that property is to be always included, independent of value of the property.
  - NON_DEFAULT: Value that indicates that only properties that have values that differ from default settings (meaning values they have when Bean is constructed with its no-arguments constructor) are to be included.
  - NON_EMPTY: Value that indicates that only properties that have values that values that are null or what is considered empty are not to be included.
  - NON_NULL: Value that indicates that only properties with non-null values are to be included.

- @JsonIgnore: simple annotation to use for ignoring specified properties, can be used in a field or a method.

- @JsonFormat: General-purpose annotation used for configuring details of how values of properties are to be serialized. Unlike most other Jackson annotations, it does not have specific universal interpretation: instead, effect depends on datatype of property being annotated. Common uses include choosing between alternate representations.

  In this project it has been used to configure how Dates are going to be parsed. For a Date attribute the possible values for their “shape” element are:

  - JsonFormat.Shape.NUMBER: Date is to be serialized as number (Java timestamp)

- JsonFormat.Shape.STRING : Date is to be serialized as a String (such as ISO-8601 compatible time value). When this mode is used the “timezone” element can be set to configure the time zone of the serialized time.

5.6.1.1 Declarative Hyperlinking annotations

Following the principle of RESTful APIs must be hypertext-driven. JAX-RS currently offers Jersey adds an annotation-based method to help with the generation of links between resources.

The annotation @Ref is used for these purposes. Links are added to representations using the @Ref annotation over URI entity class fields. The Jersey runtime is responsible for injecting the appropriate URI into the URI field prior to serialization.

Referenced or literal templates may contain parameters. Two forms of parameters are supported:

- URI template parameters, e.g. @Ref(“patterns/{id}”) where {id} represents a variable part of the URI.
- EL (expression language) expressions, e.g. @Ref(“patterns/${instance.id}”) where ${instance.id} is an EL expression.

Parameter values can be extracted from three implicit beans:

- instance: Represents the instance of the class that contains the annotated field.
- entity: Represents the entity class instance returned by the resource method.
- resource: Represents the resource class instance that returned the entity.

The source for URI template parameter values can be changed using the @Binding annotation:

```
@Ref(value="schemas/{schemaid}/classifiers/{id}",bindings={@Binding(name="
schemaid", value="${entity.id}")})
```

If an absolute URI is desired instead of an absolute path then the annotation can be changed configuring its “style” element to the to the “ABSOLUTE” value: `@Ref(value=“patterns/{id}”,style=ABSOLUTE)`

5.6.2 DTO classes implementation

In this section a detailed description of DTO classes contained in the package edu.upc.gessi.rptool.dtos and its hierarchy (Figure 5.4) will be described.

All the DTO classes have been implemented following the representation of resources design described in section 4.6 (Resources representation design).

Since most of the representations can be partial or complete the decision of using the same DTO class to implement both representations has been taken in order to do not increase unnecessarily the number of classes and the complexity of the hierarchy. This allows to reuse the same DTO class for both representations allowing, if it necessary in a future, to refactor only one class in order to change the representation of a resource.

The method that has been used to have to different representations for the same DTO class is to set the fields that are not present in the partial representation to a null value configuring the DTO classes in order to do not serialize these fields when their value is null using the @JsonInclude(Include.NON_NULL) annotation.

Also, some fields of the DTO classes are never serialized since they are only used to compute some other serialized fields such as the id, this fields have been configured using the @JsonIgnore annotation.

In order to not extend this section unnecessarily, the getters and setters of all the DTO classes are going to be ignored since they only perform the standard behaviour.

In every class the constructor is the responsible of initialize all the fields either receiving the related business class or receiving the complete set of values for each field.

5.6.2.1 DTO classes diagram

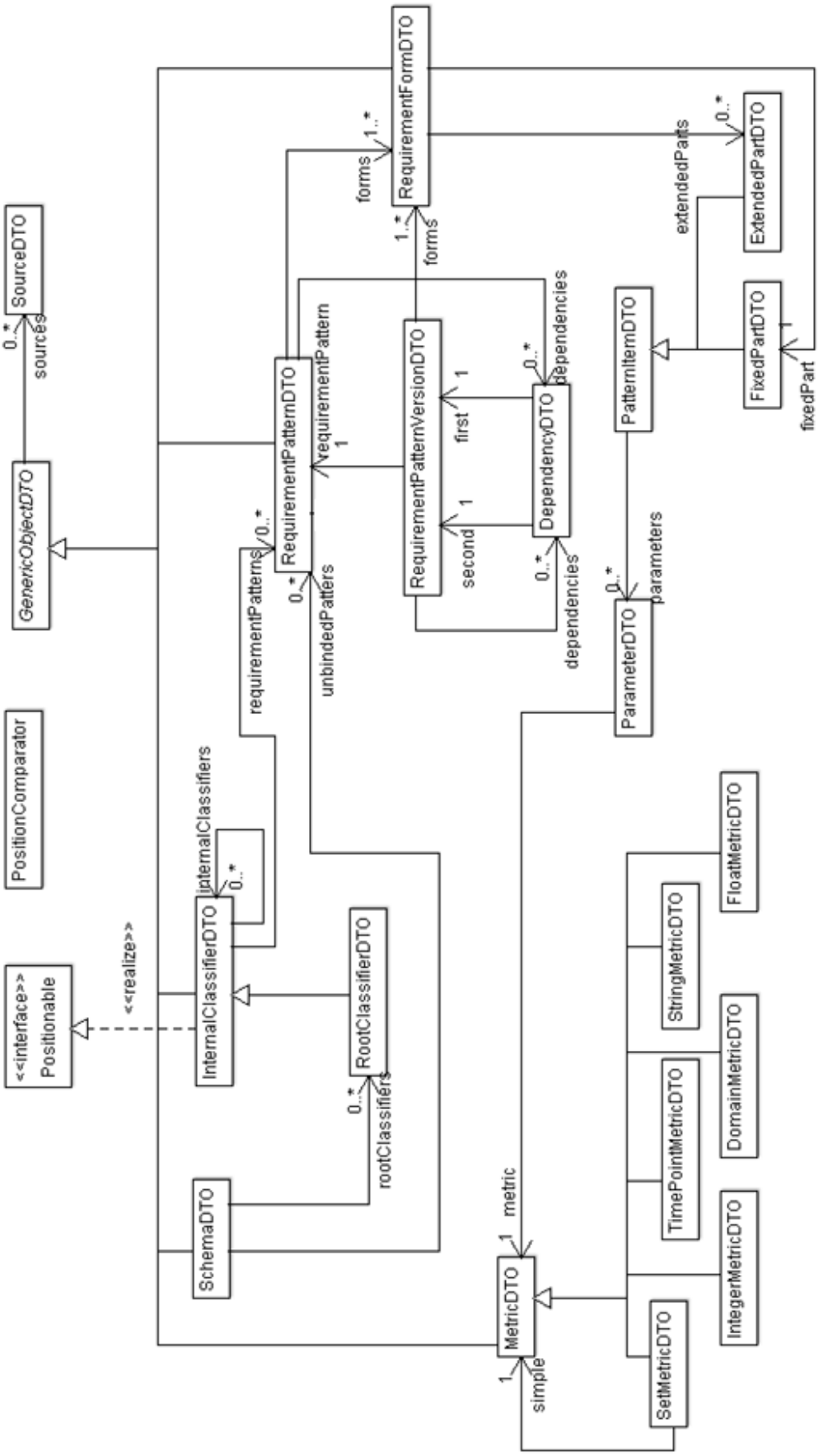


Figure 5.4. DTOs class hierarchy

5.6.2.2 GenericObjectDTO

This abstract DTO class (Figure 5.5) is implemented to contain all the common fields that must be present in the classes that extends the GenericObject business class.

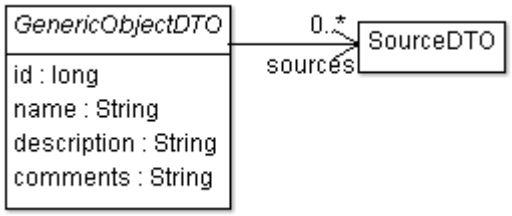


Figure 5.5. GenericObjectDTO class and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public abstract class GenericObjectDTO {
  @JsonIgnore private long id;
  private String name;
  private String description;
  private String comments;
  private Set<SourceDTO> sources;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       name: "Acceptance Tests",
       description: "This pattern expresses the need of setting the type of
       tests the customer shall perform to accept the system implementation. ",
       comments: "",
+      sources: [ ],
    }
```

5.6.2.3 SourceDTO

This DTO class (Figure 5.6) is implemented to contain all the fields that must be present in a complete or partial Source representation described in sections 4.6.7 (Individual Source) and 4.6.8 (Sources collection).

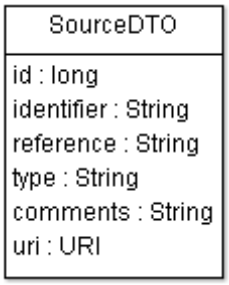


Figure 5.6. SourceDTO class and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class SourceDTO {
  @Ref(value="sources/{id}", style=Style.ABSOLUTE) private URI uri;
  @JsonIgnore private long id;
  private String identifier;
  private String reference;
  private String type;
  private String comments;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   uri: "http://localhost:8080/Pabre-WS/api/sources/786433",
   identifier: "ISO/IEC 9126-1",
   reference: "",
   type: "",
   comments: ""
}
```

5.6.2.4 RequirementPatternDTO

This DTO class (Figure 5.7) is implemented to contain all the fields that must be present in a complete or partial Requirement Pattern representation described in sections 4.6.1 (Individual pattern) and 4.6.2 (Patterns collection)

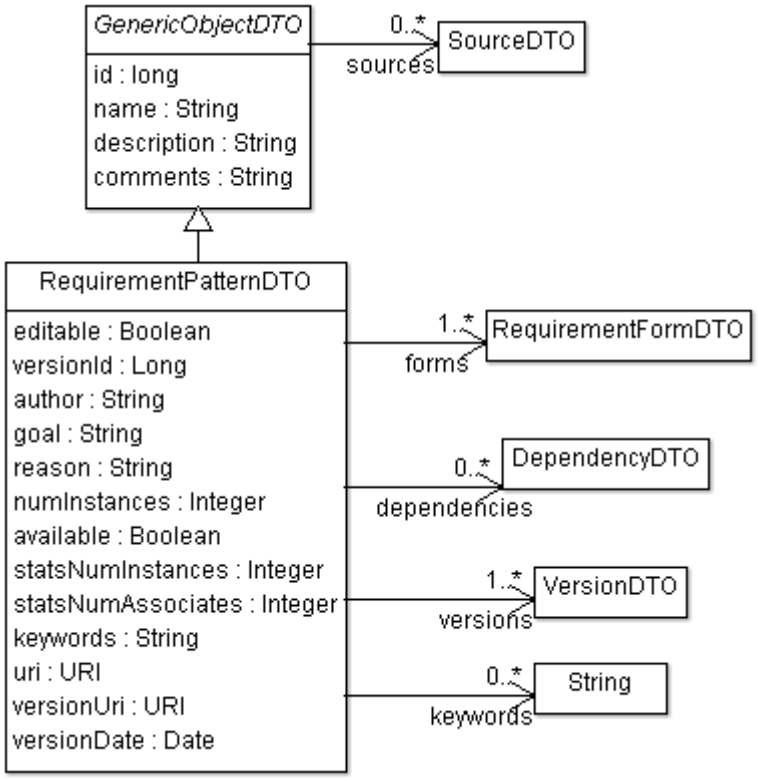


Figure 5.7. RequirementPatternDTO class and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class RequirementPatternDTO extends GenericObjectDTO {
  private String author;
  private Boolean available;
  private Set<DependencyDTO> dependencies;
  private Boolean editable;
  private Set<RequirementFormDTO> forms;
  private String goal;
  private Set<String> keywords;
  private Integer numInstances;
  private String reason;
  private Integer statsNumAssociates;
  private Integer statsNumInstances;
  private Set<RequirementPatternVersionDTO> versions


  @Ref(value="patterns/{id}", style=Style.ABSOLUTE)
  private URI uri;

  @JsonFormat(shape=JsonFormat.Shape.STRING, timezone="UTC")
  private Date versionDate;

  @JsonIgnore private Long versionId;

@Ref(value="patterns/{id}/versions/{versionId}",style=Style.ABSOLUTE)
  private URI versionUri;
}
```

And finally, and example of JSON serialization of this DTO class is shown:


```
    {
       name: "Acceptance Tests",
       description: "This pattern expresses the need of setting the type of
       tests the customer shall perform to accept the system implementation. ",
       comments: "",
+      sources: [ ],
       author: "Cristina Palomares, Samuel Renault, Carme Quer, Cindy
       Guerlain",
       available: true,
+      dependencies: [ ],
       editable: true,
+      forms: [ ],
       goal: "Stating the type of tests for the system implementation
       acceptance",
+      keywords: [ ],
       numInstances: 0,
       reason: "There were errors in the database data",
       statsNumAssociates: 0,
       statsNumInstances: 0,
       uri: "http://localhost:8080/Pabre-WS/api/patterns/20283601",
       versionDate: "2013-06-05T10:42:24.000+0000",
+      versions: [ ],
       versionUri: "http://localhost:8080/Pabre-
       WS/api/patterns/20283601/versions/1212416"
    }
```


5.6.2.5 RequirementPatternVersionDTO

This DTO class (Figure 5.8) is implemented to contain all the fields that must be present in a complete or partial Pattern Version representation described in sections 4.6.3 (Individual pattern version) and 4.6.4 (Pattern versions collection).

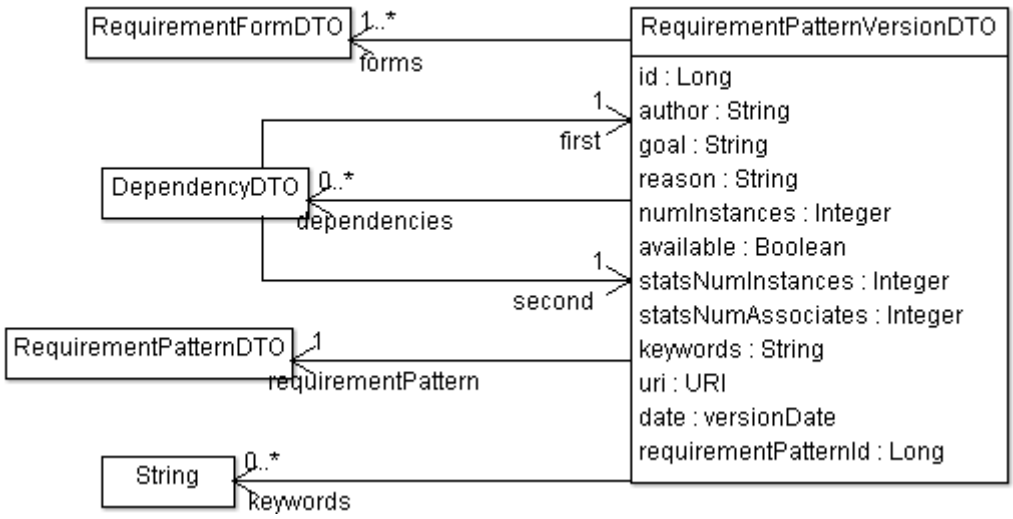


Figure 5.8. RequirementPatternVersionDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class RequirementPatternVersionDTO {
  @JsonIgnore     private Long id;
  @JsonIgnore     private Long requirementPatternId;
  private String author;
  private String goal;
  private String reason;
  private Integer numInstances;
  private Boolean available;
  private Integer statsNumInstances;
  private Integer statsNumAssociates;
  private Set<RequirementFormDTO> forms;
  private Set<String> keywords;
  private Set<DependencyDTO> dependencies;
  private RequirementPatternDTO requirementPattern;


  @JsonFormat(shape=JsonFormat.Shape.STRING, timezone="UTC")
  private Date versionDate;

  @Ref(value="patterns/{requirementPatternId}/versions/{id}",
style=Style.ABSOLUTE)
  private URI uri;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       uri: "http://localhost:8080/Pabre-
       WS/api/patterns/2031738/versions/2031738",
       versionDate: "2009-03-19T23:00:00.000+0000",
       author: "GESSI-CITI",
       goal: "Keep needed data accessible, Maintain alternative access to
       important documents in case of system failure, Be compliant with legal
       regulations ",
       numInstances: 0,
       available: true,
       statsNumInstances: 0,
       statsNumAssociates: 0,
+      forms: [ ],
+      keywords: [ ],
+      dependencies: [ ],
+      requirementPattern: { }
    }
```

5.6.2.6 RequirementFormDTO

This DTO class (Figure 5.9Figure 5.8) is implemented to contain all the fields that must be present in a complete or partial Requirement Form representation described in section 4.6.1.3 (Requirement Form complete representation).

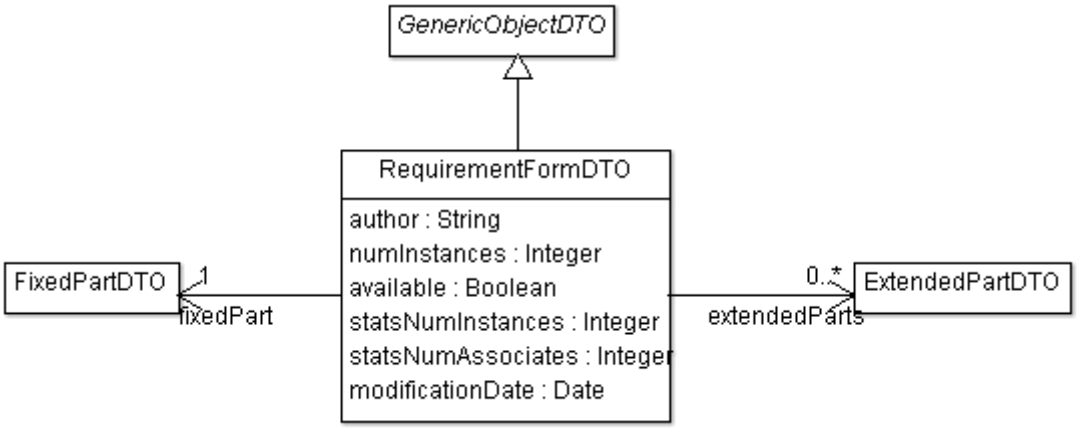


Figure 5.9. RequirementFormDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class RequirementFormDTO extends GenericObjectDTO {
  private String author;
  private Integer numInstances;
  private Boolean available;
  private Integer statsNumInstances;
  private Integer statsNumAssociates;
  private FixedPartDTO fixedPart;
  private Set<ExtendedPartDTO> extendedParts;

  @JsonFormat(shape=JsonFormat.Shape.STRING, timezone="UTC")
  private Date modificationDate;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       name: "Filing of Data",
       description: "This form expresses the general concept of filing,
       requiring to specify which type of data to file (either concrete data or
       more general description, e.g. old data, important documents, etc.). An
       extension for specifying the type of filing is also include",
       comments: "",
+      sources: [ ],
       author: "GESSI-CITI",
       modificationDate: "2009-03-19T23:00:00.000+0000",
       numInstances: 0,
       available: true,
       statsNumInstances: 0,
       statsNumAssociates: 0,
+      fixedPart: { },
+      extendedParts: [ ]
    }
```

5.6.2.7 PatternItemDTO

This abstract DTO class (Figure 5.10) is implemented to contain all the common fields that must be present in the classes that extends it, in this case, FixedPartDTO and ExtendedPartDTO described in sections 5.6.2.8 and 5.6.2.9.

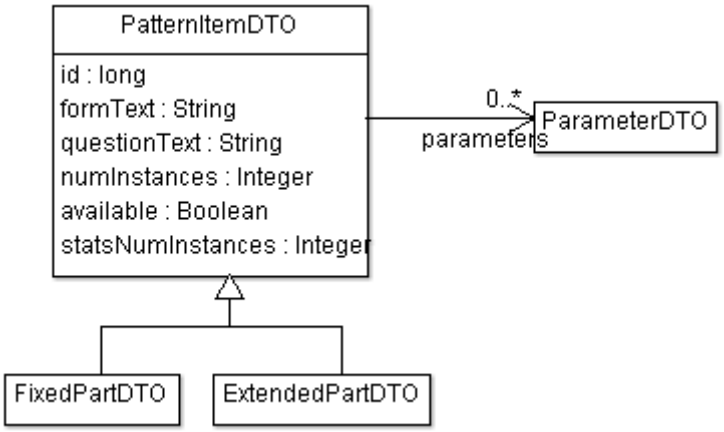


Figure 5.10. PatternItemDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class PatternItemDTO {
  @JsonIgnore private long id;
  private String formText;
  private String questionText;
  private Integer numInstances;
  private Boolean available;
  private Integer statsNumInstances;
  private Set<ParameterDTO> parameters;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       formText: "%testTypes% test design shall be based on %testBasis%.",
       questionText: "Does your company agree that %testTypes% tests willl be
       based on %testBasis%?",
       numInstances: 0,
       available: true,
       statsNumInstances: 0,
+      parameters: [ ],
    }
```

5.6.2.8 FixedPartDTO

This DTO class (Figure 5.11) is implemented to contain all the fields that must be present in a complete or partial Fixed Form representation described in section 4.6.1.4 (Fixed Part complete representation).

Figure 5.11. FixedPartDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class FixedPartDTO extends PatternItemDTO{

}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
        formText: "%testTypes% test design shall be based on %testBasis%.",
        questionText: "Does your company agree that %testTypes% tests willl be
        based on %testBasis%?",
        numInstances: 0,
        available: true,
        statsNumInstances: 0,
+       parameters: [ ],
    }
```

5.6.2.9 ExtendedPartDTO

This DTO class (Figure 5.12) is implemented to contain all the fields that must be present in a complete or partial Extended Part representation described in section 4.6.1.5 (Extended Part complete representation).

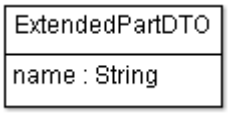


Figure 5.12. ExtendedPartDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class ExtendedPartDTO extends PatternItemDTO {
  private String name;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       formText: "%testTypes% test design shall be based on %testBasis%.",
       questionText: "Does your company agree that %testTypes% tests willl be
       based on %testBasis%?",
       numInstances: 0,
       available: true,
       statsNumInstances: 0,
+      parameters: [ ],
       name: "E4- Tests Design Background"
    }
```

5.6.2.10 DependencyDTO

This DTO class (Figure 5.13) is implemented to contain all the fields that must be present in a complete or partial Dependency representation described in section 4.6.1.1 (Dependency complete representation).

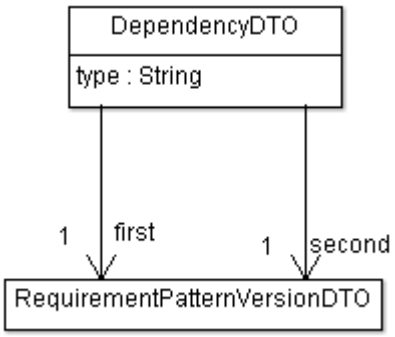


Figure 5.13. DependencyDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class DependencyDTO {
  private RequirementPatternVersionDTO first;
  private String type;
  private RequirementPatternVersionDTO second;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
+      first: { },
```

```
        type: "implies",
        second: { }
    }
```

5.6.2.11 ParameterDTO

This DTO class (Figure 5.14) is implemented to contain all the fields that must be present in a complete or partial Parameter representation described in section 4.6.1.6 (Parameter complete representation)

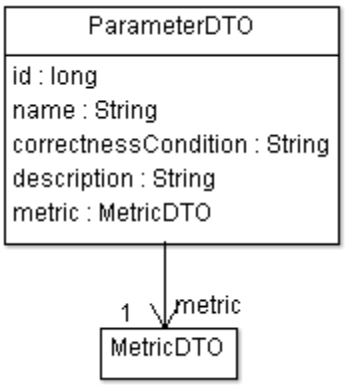


Figure 5.14. ParameterDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class ParameterDTO {
  @JsonIgnore private long id;
  private String name;
  private String correctnessCondition;
  private String description;
  private MetricDTO metric;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
        name: "testTypes",
        correctnessCondition: "",
        description: "is a non-empty set of different types of tests that can be
        done over a system implementation (e.g. verification, validation,
        etc.)",
+       metric: { }
    }
```

5.6.2.12 SchemaDTO

This DTO class (Figure 5.15) is implemented to contain all the fields that must be present in a complete or partial Schema representation described in sections 4.6.9 (Individual Schema) and 4.6.10 (Schemas collection).

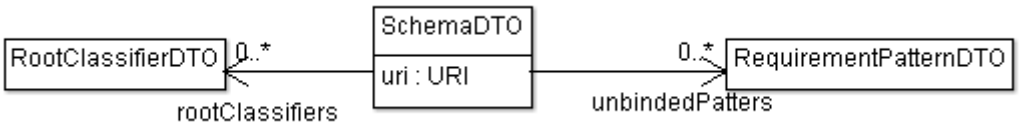


Figure 5.15. SchemaDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class SchemaDTO extends GenericObjectDTO {
  @Ref(value="schemas/{id}", style=Style.ABSOLUTE) private URI uri;
  private Set<RootClassifierDTO> rootClassifiers;
  private Set<RequirementPatternDTO> unbindedPatterns;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       name: "ISO/IEC 25010 (Extended NT, NF)",
       description: "A product quality model composed of eight characteristics
       (which are further subdivided into subcharacteristics) that relate to
       static properties of software and dynamic properties of the computer
       system. The model is applicable to both computer systems and software
       products. It has been extended with non-technical characteristics an
       subcharacteristics.",
       comments: "",
+      sources: [ ],
       uri: "http://localhost:8080/Pabre-WS/api/schemas/720897",
+      rootClassifiers: [ ],
+      unbindedPatterns: [ ]
    }
```

5.6.2.13 InternalClassifierDTO

This DTO class (Figure 5.16) is implemented to contain all the fields that must be present in a complete or partial Classifier representation described in sections 4.6.11 (Individual Classifier) and 4.6.12 (Classifiers collection).

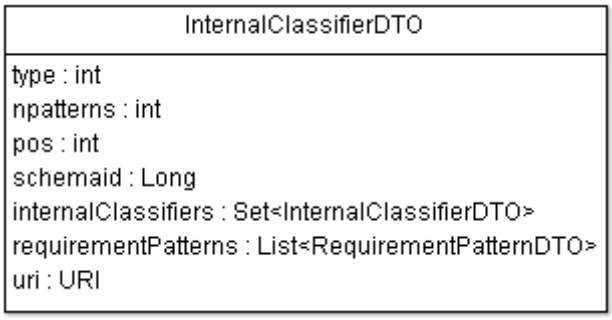


Figure 5.16. InternalClassifierDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class InternalClassifierDTO extends GenericObjectDTO implements
Positionable {
  private int type;
  private Set<InternalClassifierDTO> internalClassifiers;
  private int npatterns;
  private int pos;
  private List<RequirementPatternDTO> requirementPatterns;

  @JsonIgnore private Long schemaid;

  @Ref(value="schemas/{schemaid}/classifiers/{id}",
style=Style.ABSOLUTE)
  private URI uri;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
        name: "Non Functional",
        description: "",
        comments: "",
+       sources: [ ],
        uri: "http://localhost:8080/Pabre-
        WS/api/schemas/98330/classifiers/98331",
        type: 2,
+       internalClassifiers: [ ],
        npatterns: 0,
        pos: 0,
+       requirementPatterns: [ ]
    }
```

5.6.2.14 RootClassifierDTO

This DTO class (Figure 5.17) is implemented to contain all the fields that must be present in a complete or partial Classifier representation described in sections 4.6.11 (Individual Classifier) and 4.6.12 (Classifiers collection).

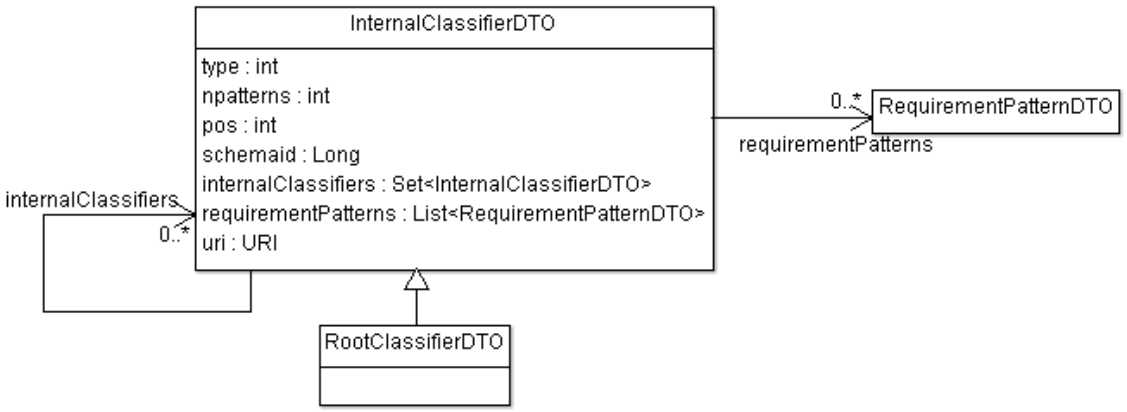


Figure 5.17. RootClassifierDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.NON_NULL)
public class RootClassifierDTO extends InternalClassifierDTO {

}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
    {
       name: "Non Functional",
       description: "",
       comments: "",
+      sources: [ ],
       uri: "http://localhost:8080/Pabre-
       WS/api/schemas/98330/classifiers/98331",
       type: 2,
+      internalClassifiers: [ ],
       npatterns: 0,
       pos: 0,
+      requirementPatterns: [ ]
    }
```

5.6.2.15 Positionable

This is not a DTO class (Figure 5.18) but it is directly related. It is a simple interface for DTOs that have a position attribute (int pos), basically classifiers, in order be able to order it using the PositionComparator class describe in section 5.6.2.16 (PositionComparator)

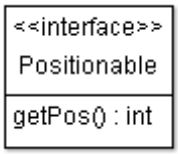


Figure 5.18. Positionable interface

Below, it is described the interface signature with the unique method it has:

```
public interface Positionable {
	public int getPos();
}
```

5.6.2.16 PositionComparator

This is not a DTO class (Figure 5.19) but it is directly related. It is a Comparator class for using it in ordered set to order Positionable objets described in section 5.6.2.15 (Positionable) by their position attribute, basically classifiers.

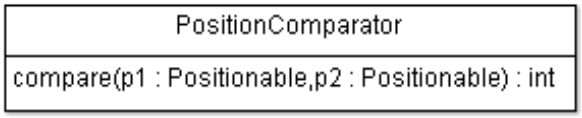


Figure 5.19. Position comparator class

Below, it is described the impletation of the class and the simple method to compare Positionable objects:

```
public class PositionComparator implements Comparator<Positionable> {
	public int compare(Positionable p1, Positionable p2) {
		return p1.getPos()-p2.getPos();
	}
}
```

5.6.2.17 MetricDTO

This DTO class (Figure 5.20) is an abstract class implemented to contain all the common fields that must be present in a complete or partial Metric representation described in section 4.6.5 (Individual Metric).

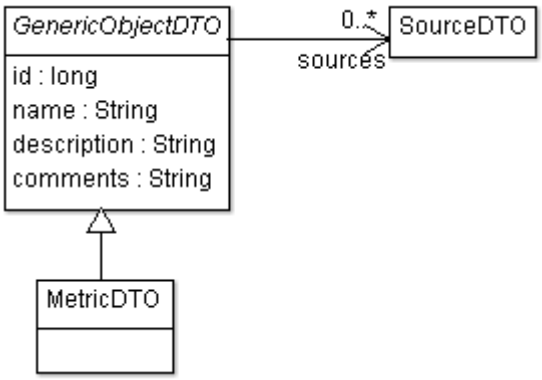


Figure 5.20. MetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
public class MetricDTO extends GenericObjectDTO {
  @Ref(value="metrics/{id}", style=Style.ABSOLUTE) private URI uri;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "UptimeInstant",
   description: "UptimeInstant = TimePoint",
   comments: "",
+  sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/2031754",
}
```

5.6.2.18 IntegerMetricDTO

This DTO class (Figure 5.21) is implemented to contain all the fields that must be present in a complete or partial Integer Metric representation described in section 4.6.5.1 (Integer Metric complete representation).

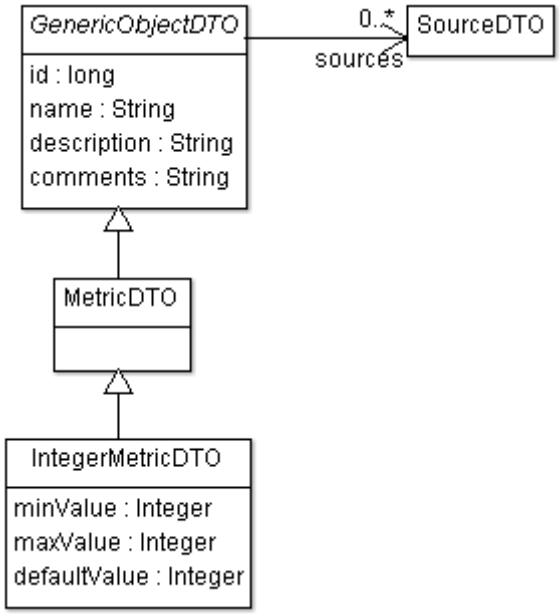


Figure 5.21. IntegerMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class IntegerMetricDTO extends MetricDTO{
  public Integer minValue;
  public Integer maxValue;
  public Integer defaultValue;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
    name: "Amount",
    description: "Amount = Integer",
    comments: "",
+   sources: [ ],
    uri: "http://localhost:8080/Pabre-WS/api/metrics/2031777",
    minValue: 0,
    maxValue: 9999999,
    defaultValue: 0
}
```

5.6.2.19 FloatMetricDTO

This DTO class (Figure 5.22 is implemented to contain all the fields that must be present in a complete or partial Float Metric representation described in section 4.6.5.2 (Float Metric complete representation).

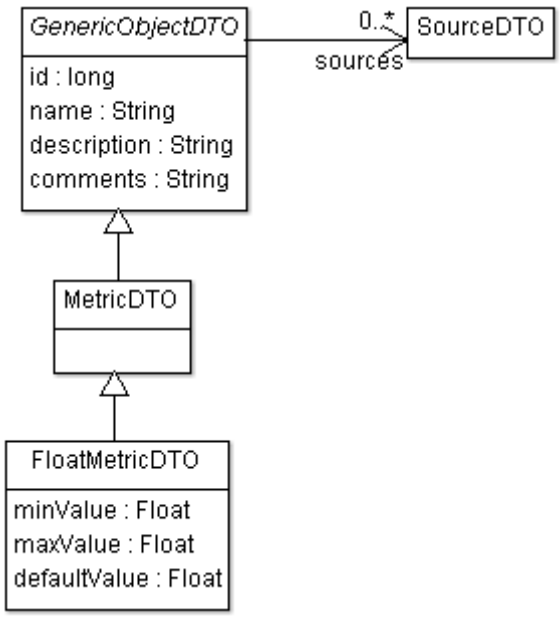


Figure 5.22. FloatMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class FloatMetricDTO extends MetricDTO {
  public Float minValue;
  public Float maxValue;
  public Float defaultValue;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "Quantity",
   description: "Quantity = Float",
   comments: "",
+  sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/20283456",
   minValue: 1.4e-45,
   maxValue: 3.4028235e+38,
   defaultValue: 3.4028235e+38
}
```

5.6.2.20 StringMetricDTO

This DTO class (Figure 5.23) is implemented to contain all the fields that must be present in a complete or partial String Metric representation described in section 4.6.5.3 (String Metric complete representation).

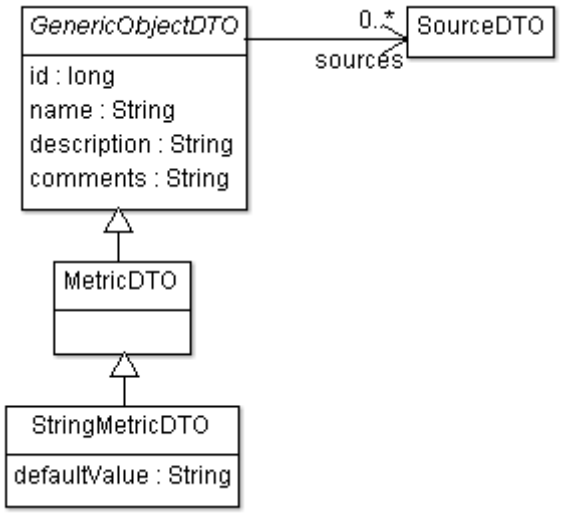


Figure 5.23. StringMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class StringMetricDTO extends MetricDTO {
  String defaultValue;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "ProjectDataType",
   description: "ProjectDataType = String. (eg. "customers data", "project
   data", ?orders data?, ?)",
   comments: "",
+  sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/20283463",
   defaultValue: null
}
```

5.6.2.21 TimePointMetricDTO

This DTO class (Figure 5.24) is implemented to contain all the fields that must be present in a complete or partial TimePoint Metric representation described in section 4.6.5.4 (TimePoint Metric complete representation).

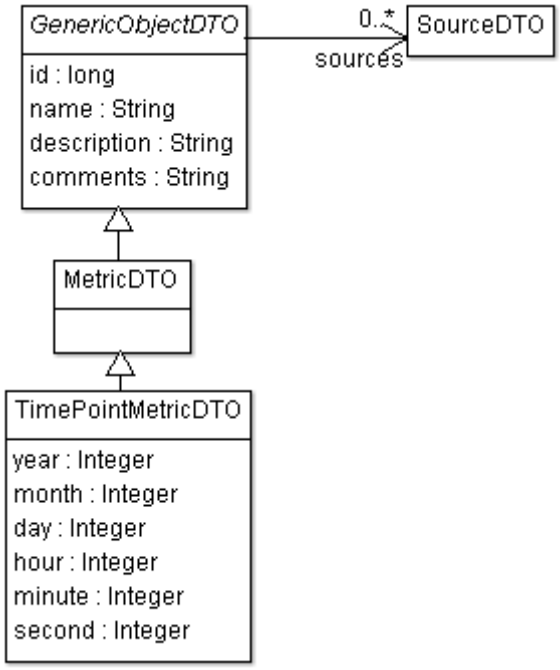


Figure 5.24. TimePointMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class TimePointMetricDTO extends MetricDTO{
  private Integer year;
  private Integer month;
  private Integer day;
  private Integer hour;
  private Integer minute;
  private Integer second;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "Date",
   description: "Date = TimePoint",
   comments: "",
+  sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/20283450",
   year: 2012,
   month: 6,
   day: 1,
   hour: null,
   minute: null,
   second: null
}
```

5.6.2.22 DomainMetricDTO

This DTO class (Figure 5.25) is implemented to contain all the fields that must be present in a complete or partial Domain Metric representation described in section 4.6.5.5 (Domain Metric complete representation).

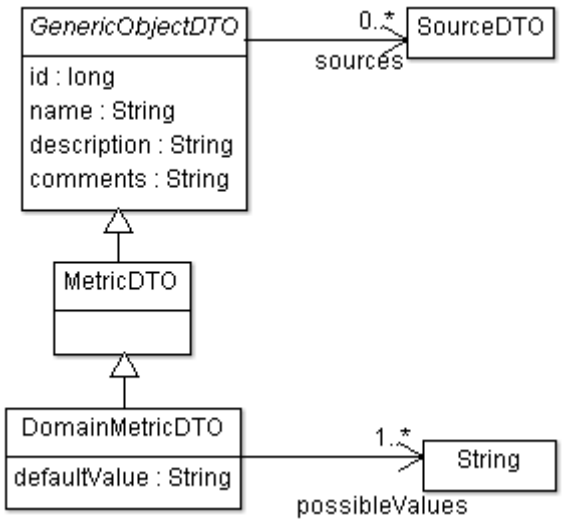


Figure 5.25. DomainMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class DomainMetricDTO extends MetricDTO{
	private List<String> possibleValues;
	private String defaultValue;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "AgentType",
   description: "AgentType = Domain(person who draft the proposal,
   employees of each party, parties subcontractors, etc.)",
   comments: "",
+  sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/20283433",
   possibleValues:
   [
          "parties subcontractors",
          "person who draft the proposal",
          "employees of each party"
   ],
   defaultValue: null
}
```

5.6.2.23 SetMetricDTO

This DTO class (Figure 5.26) is implemented to contain all the fields that must be present in a complete or partial Set Metric representation described in section 4.6.5.6 (Set Metric complete representation).

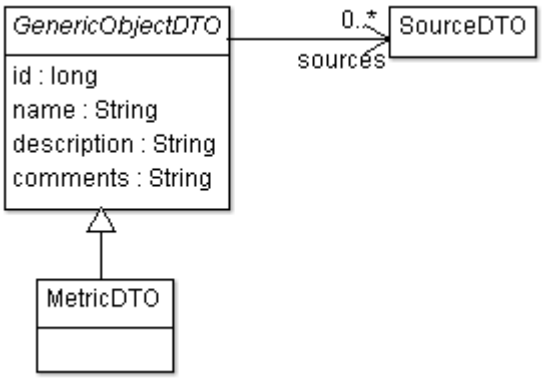


Figure 5.26. SetMetricDTO and relationships

Below, it is described the declaration of attributes using the proper annotations described in section 5.6.1 (Used annotations):

```
@JsonInclude(Include.ALWAYS)
public class SetMetricDTO extends MetricDTO {
	private MetricDTO simple;
}
```

And finally, and example of JSON serialization of this DTO class is shown:

```
{
   name: "TestTypes",
   description: "TestTypes = Set(TestType)",
   comments: "",
   sources: [ ],
   uri: "http://localhost:8080/Pabre-WS/api/metrics/20283525",
+  simple: {}
}
```

5.7 Servlet context listener class

Servlet context listener classes are classes that implement ServletContextListener interface and allows to execute some code on application deploy and undeploy through their methods contextInitialized() and contextDestroyed().

It has been detected a set of needs that require to execute code on application load and unload, so it has been implemented the

edu.upc.gessi.rptool.listeners.ResourceManagerListener class that is responsible of the following tasks:

- Application deploy tasks:
  - Log4j configuration: log4j must be initialized on application deploy loading its configuration file.
  - pabreWSRootPath system variable initialization: It has be decided to store log4j logs files into Pabre-WS application path and not in servlet container application logs folder in order to have full access to them. But, there is no way to tell log4j configuration file where to store logs since paths are relative to servlet container application. The only method it has been found to refer Pabre-WS application path is to initialize a "pabreWSRootPath" system property on application deploy and refer it as "${pabreWSRootPath" from log4j configuration file.
  - Initialize MediatorGeneric Hibernate SesionFactory: In previous PABRE tools Hibernate SesionFactory is initialized on first Hibernate request. But in order to wait until first request it has been decided to initialize it on application deploy.
- Application unload tasks:
  - Free resources: It has been detected memory leaks when redeploying the web service if some resources are not manually freed. Concretely, MediatorGenereic SesionFactory, that also frees all the existing Hibernate sessions when it is closed, and JDBC drivers that need to be deregistered.

Servlet context listener classes need to be registered in the /src/main/webapp/WEB-INF/web.xml file adding the following lines into "web-app" element:

```
<listener>
  <listener-class>
      edu.upc.gessi.rptool.listeners.ResourceManagerListener
  </listener-class>
</listener>
```

5.8 Data layer and Hibernate modifications

This section describes the necessary modifications that have been performed in hibernate configuration files and classes in order to implement the web service.ç

5.8.1 Hibernate's configuration file mapping file path

Since the location of the Hibernate's mapping file "Mapeig.hbm.xml" has been modified during the copy of the resources reused from Pabre-Man as explained in section 5.3 (Reused code from Pabre-Man) from the source package edu.upc.gessi.rptool.domain to the resources folder "/src/main/resources/config", it is necessary to modify in the Hibernate's configuration files in "/src/main/resources/config/hibernate.cfg.xml" and "/src/main/resources/config/schema.cfg.xml" the mapping resource XML element to the new path from:

```
<mapping resource="edu/upc/gessi/rptool/domain/Mapeig.hbm.xml"/>
```

To the new Hibernate's mapping file path:

```
<mapping resource="config/Mapeig.hbm.xml"/>
```

5.8.2 Hibernate's configuration file load

The class edu.upc.gessi.rptool.data.MediatorGeneric.java is the class responsible of load hibernate configuration file resource "config/hibernate.cfg.xml".

But the implementation loads the configuration file using this implementation:

```
cfg.configure(new File("config/schema.cfg.xml"));
```

And this tries to load the file from the root web application folder (/config/schema.cfg.xml) while the file in the web application is stored in a different path (WEB-INF/classes/config/schema.cfg.xml path).

In order to fix this, this line have been modified to load the configuration file using a String as a parameter in place of a File

```
cfg.configure("config/schema.cfg.xml");
```

This tries to load the file from the classes or resources folder of the web application ("WEB-INF/classes") finding it correctly.

5.8.3 Lazy property

Lazy fetching is a Hibernate 3 feature that allows to load associations of a class only when they are requested, this means that if a class is associated with a collection, the query to de database only will load the class and then in the moment when the relationships is requested, then Hibernate will automatically load the associated class. But access to a lazy association outside of the context of an open Hibernate session will result in an exception and all the mediator classes to communicate on the data layer are implemented making a single request to the database per session.

In this project I have find some requests of the web service that need to load associated classes that were not being lazy loaded when requesting the database and exception was generated when accessing it.

In order to allow these association accesses this properties of the Hibernate mapping file needed edu.upc.gessi.rptool.domain.Mapeig.hbm.xml to be modified adding the property lazy="false":

- Line 17 and 18 in section related to mapping of Dependency class:

```
<class name="edu.upc.gessi.rptool.domain.requirementPatterns.Dependency" table="dependency">
        …
        <many-to-one name="first" column="FIRST_1" not-null="true" lazy="false"/>
        <many-to-one name="second" column="SECOND_2" not-null="true" lazy="false"/>
</class>
```

- Line 105 in section related to mapping of RequirementPatternVersion class:

```
<class name="edu.upc.gessi.rptool.domain.requirementPatterns.RequirementPatternVersion"
table="REQUIREMENT_VERSION">
        …
        <many-to-one name="myRequirementPattern" column="MY_REQ_PATTERN" not-null="false"
        lazy="false"/>
</class>
```

By default, Hibernate3 uses lazy select fetching for collections and lazy proxy fetching for single-valued associations. These defaults make sense for most associations in the majority of applications.

5.8.4 MediatorPatterns class methods

Three new methods have been implemented in this data Mediator to allow specific queries to the database:

- A method to get the list of requirement patterns in which their last version has the specified keyword (or a part of the keyword) or the requirement pattern name contains a part of the keyword and returns the list of the ids, names, availability and last version editability of those patterns

```
public static List
listPatternIdWithNameEditableAvailableByKeyword(String keyword)
```

This method uses the following HQL query:

```
select p.id, p.name, p.editable, v.available
from RequirementPattern as p
  left join p.myVersions as v
  left join v.myKeywords as k
where (lower(k.name) like :keyword or lower(p.name) like
:keyword) and v.versionDate = (select max(v2.versionDate) from
p.myVersions as v2)
```

It is important to remark that keyword parameter is set as: `.setParameter("keyword","%"+keyword.toLowerCase()+"%")`

This means that the used search pattern for keyword is “%keyword%”. The pattern “%” matches any number of characters, even zero characters. Using this search pattern cannot use database possible indexes in keyword column since standard indexes cannot be used when the search pattern begins with “%”. The alternative is to create a Full-text index and perform a full-text search using MySQL MATCH() function and this search will have a much better performance. But, on the other hand, full-text indexes can be used only with MyISAM tables.

The major deficiency of MyISAM is the absence of transactions support. Also, foreign keys are not supported and in normal use cases, InnoDB seems to

be faster than MyISAM. Versions of MySQL 5.5 and greater have switched to the InnoDB engine to ensure referential integrity constraints, and higher concurrency.

Taking into account the cardinality of the keyword table, its estimated future growth, the current performance of this query and the disadvantages of using MyISAM, it has been decided agreeing with the project client, to use InnoDB storage engine and discard the use of full-text indexes despite its performance improving.

Some solution that could be developed in future work is to use an enterprise search platform as an indexing software such as Apache Solr or ElasticSearch to allow advanced searches over more requirement patterns fields and improve the performance.

- A method to get the list of the ids, names, availability and last version editability of all the patterns of the system

```
public static List listPatternIdWithNameEditableAvailable()
```

This method uses the following HQL query:

```
select p.id, p.name, p.editable, v.available
from RequirementPattern as p left join p.myVersions as v
where v.versionDate = (select max(v2.versionDate) from
p.myVersions v2)
```

- A method to get the list of the ids, names, availability, last version editability and id of the classifier where it is located of all the patterns of the system

```
public static List
listPatternIdWithNameEditableAvailableAndParents()
```

This method uses the following HQL query:

```
select p.id, p.name, p.editable, v.available, b.id
```

```
from RequirementPattern as p left join p.myVersions as v left
join p.myBasicClassifiers as b
where v.versionDate = (select max(v2.versionDate) from
p.myVersions v2)
```

5.9 Project build and package

Since Apache Maven has been used to manage the dependencies and manage the application lifecycle, project building is very easy.

It is only necessary to be placed in project root folder and use the following command:

```
$ mvn package
```

All the dependencies will be solved, the classes will be compiled and the application will be packaged in “Pabre-WS.war” file placed in “/target” folder.

6 Database migration

Aside of developing the web services the project had an extra goal that consists on migrating the current databases of PABRE tools.

The motivation of this goal is to allow the three software tools in the PABRE framework to be compatible with MySQL DBSM in order store the SRP and projects catalogue in a persistent database is already deployed in the project client infrastructure and allow the requirements engineer expert that manages the catalogue to work directly in a centralized database in place of a local Apache Derby database or even a centralized Apache Derby since MySQL has better permanent availability support in the client infrastructure.

The DBMS migration process is composed by three phases:

- Create a new MySQL database.
- Purpose an automatic method to perform the schema and data migration from Apache Derby to MySQL and vice versa.
- Adapt the code of the tools to be compatible with the new DBMS (MySQL) while still remains compatible with the old one (Apache Derby) making minimum configuration modifications.

6.1 MySQL database creation

Two databases have been created in MySQL DBMS using the latin1_general_cs:

```
CREATE DATABASE pabre_patrons
CHARACTER SET latin1 COLLATE latin1_general_cs;

CREATE DATABASE pabre_projectes
CHARACTER SET latin1 COLLATE latin1_general_cs;
```

The collation latin1_general_cs is have been selected in order to avoid conflicts with columns that have the unique flag and have rows with keys that differ only in their letter case. This is possible since the default Apache Derby collation used in the database of this project is case sensitive and can produce duplicated primary keys in migration process if a non-case sensitive collation is selected.

Once databases are created, two different users, with different passwords have been created, granting each one of them, access to one of the created databases.

Finally, since no storage engine has been selected, the default storage engine in MySQL 5.6 is InnoDB and it will be used for all the tables.

6.2 Database schema and data migration

6.2.1 SQuirreL DBCopy plugin

This task was performed using the DBCopy SQuirreL plugin described in section 4.3.11 (SQuirreL SQL Client). This plugin easily allows copying database objects (schema definition and data) from one SQuirreL session window to another. The source and destination sessions can be different database vendors using Hibernate internally for data type translations.

This process is as simple of connect to both databases, select all the tables that are going to be copied from one session window and paste them in the other database session windows. But this method had some problems:

- Although the MySQL DBMS schema had a character set and collation that supported them, some non ASCII characters caused an error in the copy process that aborted it, avoiding to finish the migration completely if those characters were not modified.
- Although some dates stored in some rows where allowed in the Apache Derby and also allowed in MySQL (such as 1900-01-01), they caused an error in the copy process that aborted it, avoiding to finish the migration completely if those dates were not modified.
- Primary keys and foreign keys where not correctly migrated to MySQL database schema configuring them only with the UNIQUE attribute. Although this did not caused a problem when using the MySQL from the software tools through Hibernate, this can end up causing integrity problems in the database and when the opposite process was performed (MySQL to Apache Derby migration) it was impossible due to the primary and foreign keys information lost.

So after trying this method, it was discarded due to it necessity of making some changes in the database information and it schema information loss.

6.2.2 Apache DdlUtils

This component described in section 4.3.12 (Apache DdlUtils) allows to save the schema and data of a SQL database in a Turbine XML format, this format expresses the database schema in a database-independent way by using JDBC datatypes instead of raw SQL datatypes which are inherently database specific.

These the schema and data from a database can be fed into DdlUtils via its Ant task. DdlUtils comes with two Ant tasks that allows, insert schemas and data into the database or dump the database structure and data contained in it to XML:

- DdlToDatabaseTask: Task for performing operations on a live database. Sub tasks It has sub tasks for creating the schema in the database, drop database schemas, insert data into the database, create DTDs for data files, or write the SQL for creating a schema to a file.
- DatabaseToDdlTask: Task for getting structural info and data from a live database. It has sub tasks for writing the schema of the live database or the data currently in it to an XML file, for creating the DTDs for these data files, and for generating SQL to creating a schema in the database to a file.

Apache DdlUtils is very easy to configure, once Apache ant is installed and the JDBC drivers of the DBMSs have been be copied into the /lib folder of DdlUtils (in this case mysql-connector-java-5.1.14.jar and derbyclient.jar) the environment is prepared.

To configure the migration, an Apache Ant's XML buildfile must be written making use of the two DdlUtils tasks previously described (DdlToDabaseTask and DatabaseToDdlTask) and configuring the tasks with the DBMS connection parameters.

This Ant's buildfile has the following header:

```
<?xml version="1.0"?>
<project name="MigrateToDerby" basedir=".">

   <path id="classpath">
      <fileset dir="./lib">
         <include name="**/*.jar"/>
            <include name="*.jar"/>
```

```
      </fileset>
    </path>
    …
    [TASKS]
    …
</project>
```

To dump the schema and the data of a DBMS a DatabaseToDdlTask must be configured, this is an example of a DatabaseToDdlTask that dumps a local Derby database and store its schema in patterns-schema-xml file and its data in patterns-data.xml file:

```
<target name="import-from-derby-patterns"
description="Dumps db structure and data">
  <taskdef name="databaseToDdl"
  classname="org.apache.ddlutils.task.DatabaseToDdlTask">
    <classpath refid="classpath"/>
  </taskdef>
  <databaseToDdl modelName="MigrateTest">
    <database url="jdbc:derby://localhost:1527/database"
    driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
    username="username"
    password="password"/>
    <writeSchemaToFile outputFile="patterns-schema.xml"/>
    <writeDataToFile outputFile="patterns-data.xml"/>
  </databaseToDdl>
</target>
```

To store schemas and data of a XML file to a DBMS a DdlToDabaseTask must be configured, this is an example of a DdlToDabaseTask that stores the schema contained in patterns-schema-xml file and the data contained in in patterns-data.xml file in a local MySQL database:

```
<target name="export-to-mysql-patterns" description="Creates db and
loads data">
  <taskdef name="ddlToDatabase"
  classname="org.apache.ddlutils.task.DdlToDatabaseTask">
    <classpath refid="classpath"/>
  </taskdef>
  <ddlToDatabase schemaFile="patterns-schema.xml">
    <database url="jdbc:mysql://localhost:3306/pabre"
    driverClassName="com.mysql.jdbc.Driver"
    defaultautocommit="false"
    username="username"
    password="password"/>
    <createDatabase failonerror="false"/>
    <writeSchemaToDatabase/>
    <writeDataToDatabase datafile="patterns-data.xml"
    usebatchmode="true" batchsize="1000"/>
  </ddlToDatabase>
</target>
```

Using these both tasks either Apache Derby or MySQL database can be dumped and a schema or data XML file can be stored in either Apache Derby or MySQL database. The process has been tested in both senses and all the information about the schema and the data is preserved.

6.3 Software tools adaptations to MySQL

This sections describes the software tool modifications performed in Pabre-Man, Pabre-Proj, and Pabre-Proj-Web to adapt them to the new MySQL DBMS.

Having in account that there is an intermediate Hibernate layer between business and data layer, all the modifications are going to be performed over Hibernate configuration to allow correct communication with the DBMS.

6.3.1 Hibernate configuration file

The first modification is in Hibernate configuration file of all the software tools (/src/main/resources/config/hibernate.cfg.xml and /src/main/resources/config/schema.cfg.xml). In these files the following parameters must be modified:

```
<property name="connection.driver_class">
com.mysql.jdbc.Driver
</property>

<property name="connection.url">
jdbc:mysql://localhost:3306/pabre
</property>

<property name="connection.username">
username
</property>

<property name="connection.password">
password
</property>

<property name="dialect">
org.hibernate.dialect.MySQLDialect
</property>
```

- connection.driver_class: Is used to configurate the JDBC diver that Hibernate is going to use to connect and communicate with the database.
- connection.url: Is used to configure the url address of the database.

- connection.username: Used to configure the connection's username of the database.
- connection.password: Used to configure the connection's password of the database.
- dialect: Is used to configure the SQL dialect Hibernate is going to use to communicate the database.

The second modification is over Hibernate mapping file (/src/main/resources/config/Mapeig.hbm.xml). In this file the following modifications have been made:

6.3.2 Mapping file. Identifiers generator

The optional <generator> child element names a Java class used in the Hibernate mapping file (/src/main/resources/config/Mapeig.hbm.xml) to generate unique identifiers for instances of the persistent class. This generators are used to create the primary key identifier in database rows using autoincremental values, sequences, etc…

There are several generation algorithms for identifiers and it must be configured in Hibernate mapping file. But, the problem is that the generator configured in Hibernate mapping file was set to "native".

```
<generator class="native"/>
```

Hibernate documentation says that this generator algorithm selects "identity", "sequence" or "hilo" algorithm depending upon the capabilities of the underlying database.

This means that using Apache Derby, native will select the "hilo" algorithm, while using MySQL native selects "identity" algorithm.

Having in account that Hibernate documentation says that "identity" is supports identity columns only in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL databases and that in schema and data migration no identity column was created because Apache Derby do not support them, this algorithm must be discarded.

On the other hand, “hilo” algorithm is supported in both databases and the information needed to generate them is also correctly migrated (table “hibernate_unique_key” this is the algorithm that has been selected.

So, the modification made to fix this problem in Hibernate mapping file is to modify all the identifier generators in the file to:

```
<generator class="hilo"/>
```

6.3.3 Mapping file. Table name references

Apache Derby table names are case insensitive, so they can be referred either in lowercase or uppercase.

On the other hand, MySQL case sensitiveness for identifying tables has a different behaviour depending on the lower_case_table_names system variable. This variable has a different default value depending on the operating system:

- If lower_case_table_names is 0 database names are stored in disk using the lettercase specified when creating the table and table references are case sensitive. This is the default value on Unix/Linux systems.
- If lower_case_table_names is 1 database names are stored in disk in lowercase and table references are not case sensitive. This is the default value on Windows systems.
- If lower_case_table_names is 2 database names are stored in disk using the lettercase specified when creating the table and table references are not case sensitive. This is the default value on Mac OS X systems.

In addition, another consideration to take in account is that on Windows Windows, InnoDB storage engine always stores database and table names internally in lowercase and Apache Derby database schema for projects and patterns has all the tables are stored in uppercase.

So, having in account that Apache Derby is case insensitive and will allow references both in lowercase and uppercase, there are two possibilities to make the software compatible with any system MySQL server database.

- Set lower_case_table_names to 1 on all systems to force names to be converted to lowercase allowing case insensitive table reference from the software code.
- Manually change any table reference in the software code to reference all the tables in lowercase.

Since the first option (set lower_case_table_names to 1) on all the systems is not always possible due to administrator permissions and security reasons the second option (change table references in the software code to lowercase) has been selected.

Since all the table references are stored in the Hibernate mapping file (/src/main/resources/config/Mapeig.hbm.xml), using this method need to edit this file and change all the references to tables, stored in the "table" property of all the "class" and "joined-subclass" elements from uppercase to lowercase.

This is an example of the modification performed over of all the "class" and "joined-subclass" elements of Hibernate mapping file:

- It has been modified from a reference to "REQUIREMENT_PATTERN"

```
<class name="edu.upc.gessi.rptool.domain.requirementPatterns.Req
uirementPattern" table="requirement_pattern">
…
</class>
```

- To a reference to "requirement_pattern"

```
<class name="edu.upc.gessi.rptool.domain.requirementPatterns.Req
uirementPattern" table="requirement_pattern">
…
</class>
```

6.3.4 Hibernate JDBC connection pool

Hibernate's own connection pooling algorithm is, quite rudimentary for a multi user application. It is intended to help you developers started and is not intended for use in a production system, or even for performance testing. Hibernate recommends using a third party pool for best performance and stability.

In addition, in the context of this project several web service users must be supported at the same time and a permanent and stable connection is required with the database.

It has been found a problem with MySQL related with the default behaviour of close client connections after 8 hours of no activity.

This Hibernate's own pooling algorithm was used in the data layer of PABRE previous tools with good results because they were using Apache Derby and that DMBS was frequently used during its execution and non-permanent connection was needed like in a deployed web application.

So, when the web service does not send any communication to the MySQL database during 8 hours or the DMBS is reset, the connection is closed and the Hibernate's own connection pooling algorithm is not able to reconnect the database producing an exception.

In order to fix this problem and support a better connection pooling strategy it has been replaced the Hibernate internal connection pool by the third party C3P0 connection pooling.

C3P0 is an open source JDBC connection pool distributed along with Hibernate. Hibernate will use its org.hibernate.connection.C3P0ConnectionProvider for connection pooling setting hibernate.c3p0.* properties in its configuration file.

This connection pooling provider offers two different strategies to test the database connection aliveness in order to avoid exceptions when connecting the database.

- Method 1: Test connections always when they are retrieved from the pool just before use them, this ensures the more reliable testing strategy to assure that always the connections are alive, if they are not, it will be detected and a reconnection will be performed. This method has the disadvantage of decrease the performance of the application and overload the database since every transaction with the database will cause another previous transaction to test the connection and the number of transactions will be duplicated.
- Method 2: Connections are tested each time they are returned to the pool after a transaction, and they will be also tested periodically when they are idle. This strategy has a better performance but it has the inconvenience of

producing some exceptions if a connection is requested between the connection death and the periodical test, producing an exception that will avoid the client of the web service a success. Since the connection is tested when it is returned to the pool, the problem will be detected and the next request will be successful.

Taking in account the load of the database and the estimated future workload of the web service and the DBMS, it has been decided to use the method 1 to completely avoid web service errors and ensure a 100% of success on requests.

If the workload conditions of the web service varies in a future or the workload of the underlying DBMS is too high to support duplicated transactions the second method should be used to avoid such a demanding testing strategy.

The modifications performed over the Hibernate configuration file (/src/main/resources/config/hibernate.cfg.xml and schema.cfg.xml) to implement this connection pooling algorithm with the described testing strategy it is necessary to modify the following line:

```
<!-- JDBC connection pool (use the built-in) -->
<property name="connection.pool_size">1</property>
```

Replacing by the following lines

```
<!-- JDBC connection pool (use the c3p0) -->
<property name="connection.provider_class">
org.hibernate.connection.C3P0ConnectionProvider
</property>
<property name="hibernate.c3p0.acquire_increment">3</property>
<property name="hibernate.c3p0.max_size">100</property>
<property name="hibernate.c3p0.max_statements">0</property>
<property name="hibernate.c3p0.min_size">3</property>
<property name="hibernate.c3p0.timeout">28800</property>
<property name="hibernate.c3p0.preferredTestQuery">SELECT 1</property>
<property name="hibernate.c3p0.maxIdleTimeExcessConnections">
1200
</property>
<!-- c3p0 connection connection testing. See:
http://www.mchange.com/projects/c3p0/#configuring_connection_testing -
->
<!-- Method 1: Tested preriodically and on check-in --> <!--
<property name="hibernate.c3p0.testConnectionOnCheckin">
true
</property>
<property name="hibernate.c3p0.idle_test_period">1800</property>
```

```
-->

<!-- Method 2: Tested always on check-out -->
<property name="hibernate.c3p0.testConnectionOnCheckout">
true
</property>
<property name="hibernate.c3p0.idle_test_period">0</property>
```

In this configuration it has been selected a pool connection size of 3 to 100 connections, incrementing the pool size by 3 when there are no more available connections, discarding standard connections after 28800 idle seconds (8 hours) and the excess of connections over the minimum (3) after 1200 idle seconds (20 minutes). The test query that has been configured to test the connection is “SELECT 1;”.

7 Test client interface for Pabre-WS

In order to test the web service one goal of the project is to create a demo client that shows an example on what a client is able to using the resources that the web service offers.

It has been purposed to create a web application that allows to navigate over the SRP catalogue using different schemas and shows all the information of the last versions of requirement patterns. This web application will be designed as the current pattern catalogue that is shown on the PABRE framework web of GESSI group shown in Figure 7.1 (The GESSI group, s.f.).

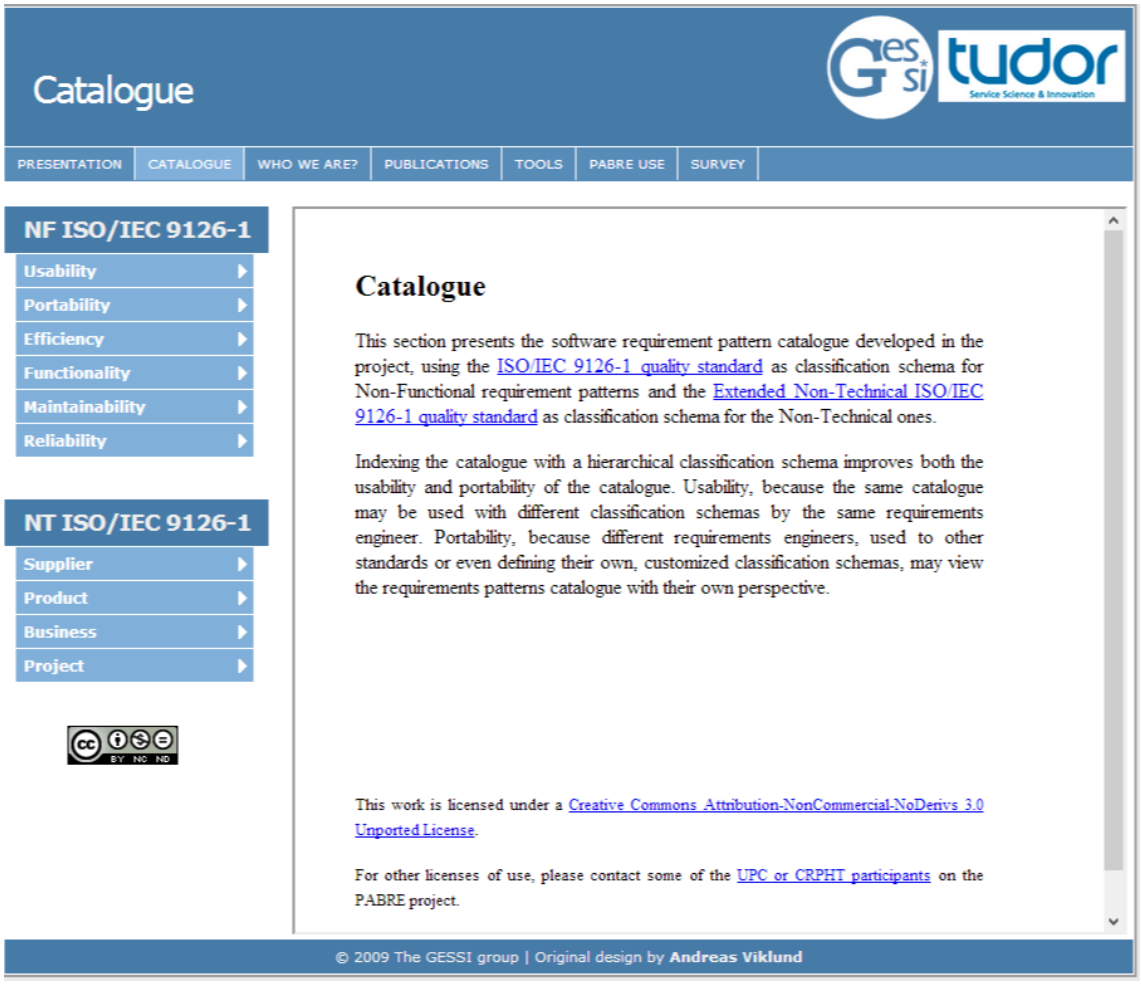


Figure 7.1. Pattern static catalogue in PABRE framework web.

This web is currently a static webpage where all the SRP of the catalogue are statically written in HTML pages, so all their information is not updated with the last version of the catalogue.

The implemented web client, will allow to navigate through the schemas, classifiers and patterns showing the information that the web service is returning from the SRP catalogue, the interface of the web client is shown in Figure 7.3.

The architecture schema of the client in shown in Figure 7.2.

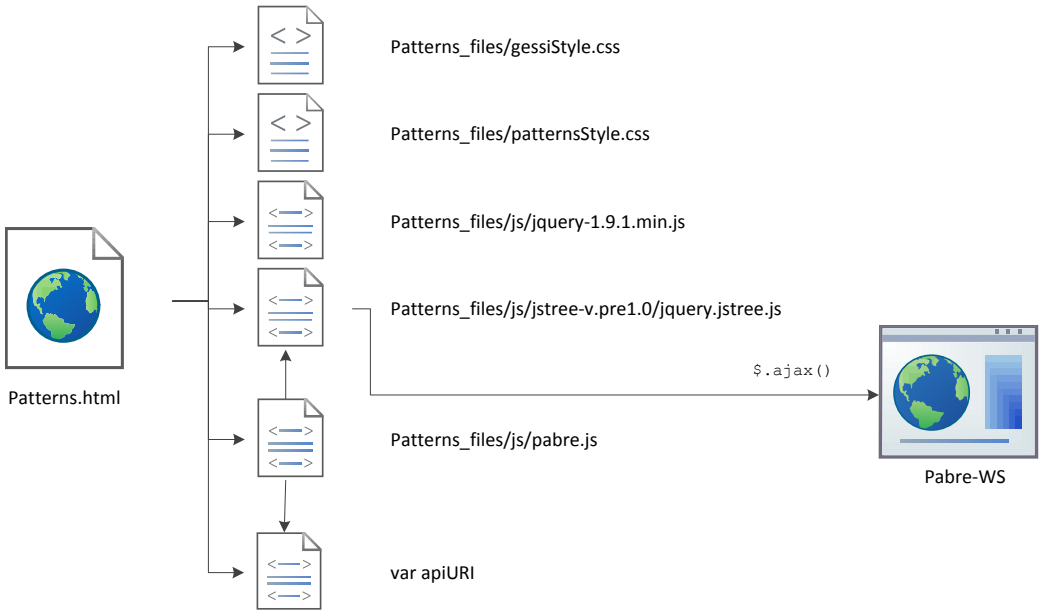


Figure 7.2. Web client architecture schema

- gessiStyle.css is the cascading style sheet used in GESSI group web, it has been used to give the client interface the same style and appearance that the GESSI group web in order to suit the client in that web once finished.

- patternStyle.css is a new cascading style sheet that contains some new declarations used in elements of the web application.

- jquery-1.9.1.min.js is a well-known multi-browser JavaScript library designed to simplify the client-side scripting of HTML. It has been used to facilitate the implementation of the web application and manage AJAX requests and responses to communicate with Pabre-WS RESTful web service.

- jquery.jstree.js: Is a JavaScript based, cross browser tree component that allows creating, manipulating and displaying a tree structure in a web page. It has been used to show classifiers and requirement patterns hierarchy of a given schema in the web application. It is packaged as a jQuery plugin.

- var apiURI is a variable initialization declared in the HTML file in order to configure the URL of the web API that is going to be used by pabre.js script:

```
var apiURI = "http://localhost:8080/Pabre-WS/api";
```

- pabre.js is the JavaScript script that initializes and controls all the behavior of the web application. It is described in the following section.

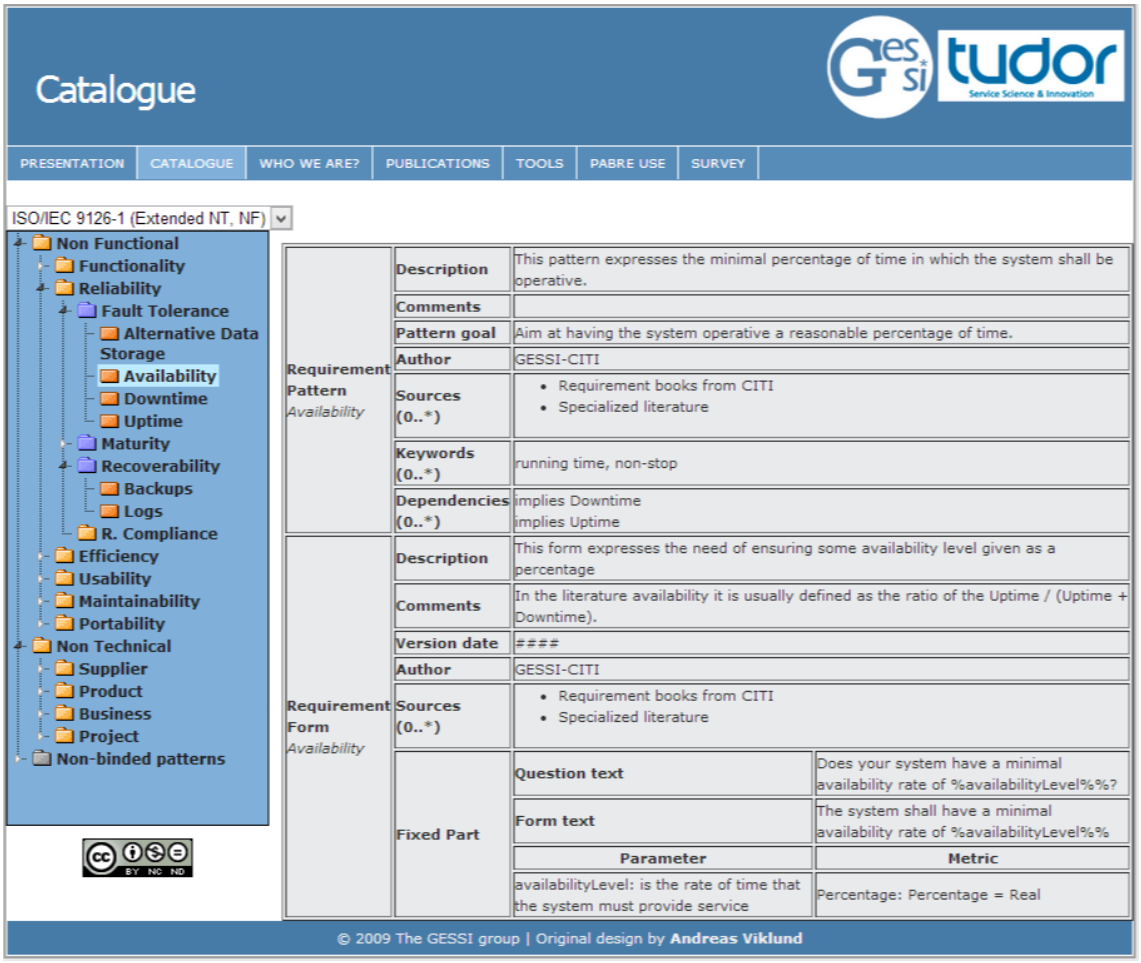


Figure 7.3. Interface of the WebApplication.

7.1 pabre.js script

pabre.js script contains the JavaScript functions that control the web client. They load the available schemas from the web service, build the tree of classifiers and patterns and show information about patterns. It has the following functions:

7.1.1 jQuery ready function

```
jQuery(function($))
```

This function is executed when the DOM is ready once the Patterns.html webpage has been loaded. It just call to initHTMLElementsState() and initHTMLElementsBehaviour() functions.

7.1.2 initHTMLElementsState function

```
function initHTMLElementsState()
```

This function initializes the state of the components of the webpage, concretely, it performs an AJAX request to Pabre-WS and processes the response, creating the combo box options with the names of the schemas of the responses and its URIs. It also adds one more option called "Alphabetical list" that references to “{apiURL}/patterns” URI in it “val” field.

7.1.3 initHTMLElementsBehaviour function

```
function initHTMLElementsBehaviour()
```

This function initializes the behaviour of the components of the webpage, concretely, it configures the top-left combo box to call loadSchema() function using the value of the selected element when its selected value changes.

7.1.4 loadSchema function

```
function loadSchema(schemaURI)
```

This function creates and configures the left tree hierarchical view using jsTree plugin. This tree has been configured to have 4 different type of nodes:

- basicClassifier: Nodes that represent a basic classifiers that directly contain one or more patterns. They are identified by a yellow folder icon.
- classifier: Nodes that represent other classifiers, such as root classifiers, empty classifiers or intermediate classifiers that do not directly contain patterns. They are identified by a blue folder icon.
- pattern: Nodes that represent requirement patterns. They are identified by an orange file icon.
- unbindedClassifier: Nodes that represent schema’s unbinded patterns classifier. They are identified by a grey folder icon.

Once the tree have been configured an AJAX request is performed to the URI received in the “schemaURI” parameter and the response is processed.

Processing the response consist in going through all the fields of the JSON object received and call converting functions described in section 7.1.5 (Converting functions) for each root classifier, requirement pattern or unbinded patterns classifier found.

Finally the tree is configured to call loadPattern() function with the URI of the selected node when a “pattern” node is selected.

7.1.5 Converting functions

```
function convertInternalClassifierToTree(internalClassifier)

function convertRequirementPatternToTree(requirementPattern)

function convertRootClassifierToTree(rootClassifier)

function convertUnbindedPatternsToTree(unbindedPatterns)
```

These functions are basically used to receive JSON objects from the Pabre-WS response and process it to create the nodes of the JSON structure that will be used to initialize jsTree and show the hierarchical tree view. They receive a JSON object and return a jsTree node. This is an example of a jsTree JSON structure used to generate the tree view:

```
[
  {
    data: { title: "Reliability" },
    attr: { rel: "classifier" },
    children:
      [
        {
          data: { title: "Fault Tolerance" },
          attr: { rel: "basicClassifier" },
          children:
            [
              {
                data: { title: "Alternative Data Storage" },
                attr:
                  {
                    rel: "pattern",
                    uri: "http://localhost:8080/Pabre-WS/api/patterns/2031738"
                  }
              }
            ]
        }
      ]
  },
```

```
  {
    data: { title: "Non-binded patterns" },
    attr: { rel: "unbindedClassifier" },
    children:
      [
        {
          data: { title: "Community Support" },
          attr:
            {
              rel: "pattern",
              uri: "http://localhost:8080/Pabre-WS/api/patterns/20283613"
            }
        },
        {
          data: { title: "Acceptance Tests" },
          attr:
            {
              rel: "pattern",
              uri: "http://localhost:8080/Pabre-WS/api/patterns/20283601"
            }
        }
      ]
  }
]
```

As it is shown each node has the following fields:

- data: Is an object that specifies the content to be load in the node. It has an “title” attribute that contains the text that will appear near the node in the tree
- attr: Is an object that contains all the attributes that will be parsed to HTML attributes of the node element. In this script is used to store the type of the node in the attribute “rel” and the uri of the patterns in the attribute “uri”.
- children: Is an array that contains the children nodes.

7.1.6 loadPattern function

This function, that is called when a pattern is selected in the patterns tree view, receives the uri of a requirement pattern, performs an AJAX request and process the response to build the table that contains all the information about the requirement pattern and will be shown on the right side of the web page such as in Figure 7.3.

8 Project plan and execution

8.1 Project planning

This section describes the time planning followed during the project describing planning of the different phases and the estimated time.

The project started on February, 11th 2013 when it was inscribed. The project was planned from the beginning to develop it during one semester and finalize and expose it on September, 2013.

Since it was the only subject I was enrolled during that semester I planned to develop it in full-time. So an average of 8 hours of work per day was planned for the project plan.

The project plan was divided in X tasks in order to assign them an estimated time cost based on the project goals:

1. State of the art about web services.

   This phase consist in studying and documenting about the different web service technologies, protocols, architectures, principles and features that nowadays web services use and offer.

2. Analysis of the current tools of PABRE framework.

   Since the development of the web service was based on the already developed PABRE framework tools, it was need to study the architecture, the domain, the data layer, the code, and the features of these tools in order to decide which resources can be reused, what level of compatibility can be reached and what alternatives are suitable for database migration.

3. Study of the available tools and environment to develop the project.

   Once the architecture and the style of the web service has been decided it was needed to study what environment tools, libraries and software is suitable to develop the project.

4. Project environment and tools learning, installation and configuration.

Since, most of the tools were never used previously by me and I did not have a deep knowledge about them, a period of time is needed to study the documentation, read tutorials, familiarize with them, install, configure and make them work.

5. Web service design.

This phase is dedicated to the design of the web service, its architecture, its structure, the design of the classes and the features it will offer.

6. Web service implementation.

This phase is dedicate to the implementation of the web service, the modifications that are needed over the reused code, the implementation of the new classes, solve the problems that will appear during the development and test the new implemented features during the development.

7. Test client implementation.

In this phase the test client that makes use of the web service to access the SRP catalogue is designed, developed and tested.

8. Database migration.

During this phase it will be studied the different available alternatives to purpose a semi automatized method to perform database migration between both DBMSs and the needed modification in the software tools to make them compatible with both of them.

9. API documentation.

In this section the API documentation for clients and developers of tools that make use of the web service is elaborated in order to allow them accessing it.

10. Final report elaboration

This task consist in the elaboration of the final report document that contains all the documentation related with the project.

The estimated planned time for each task is described in Figure 8.1.

Task	Task description	Estimated time
1	State of the art about web services.	42 h
2	Analysis of the current tools of PABRE framework.	70 h
3	Study of the available environment tools to develop the project.	21 h
4	Project environment and tools learning, installation and configuration.	84 h
5	Web service design.	70 h
6	Web service implementation.	210 h
7	Test client implementation.	56 h
8	Database migration.	35 h
9	API documentation.	20 h
10	Final report elaboration	150 h
	TOTAL	**758 h**

Figure 8.1. Project planning tasks time

8.2 Project execution

The calendar in Figure 8.3 shows the period of time placed between 11th February, 2013 the date when the project started and 15th September, 2013 the day when the project and the documentation finished. It is the period of time when this project has been realized.

The days are coloured using different backgrounds depending on the average number of hours of dedication. The meaning of the colours is described in Figure 8.2.

	Average dedication of 7 hours per day
	Average dedication of 4 hours per day
	Average dedication of 9 hours per day
	Day not dedicated to the project

Figure 8.2. Calendar colours meaning

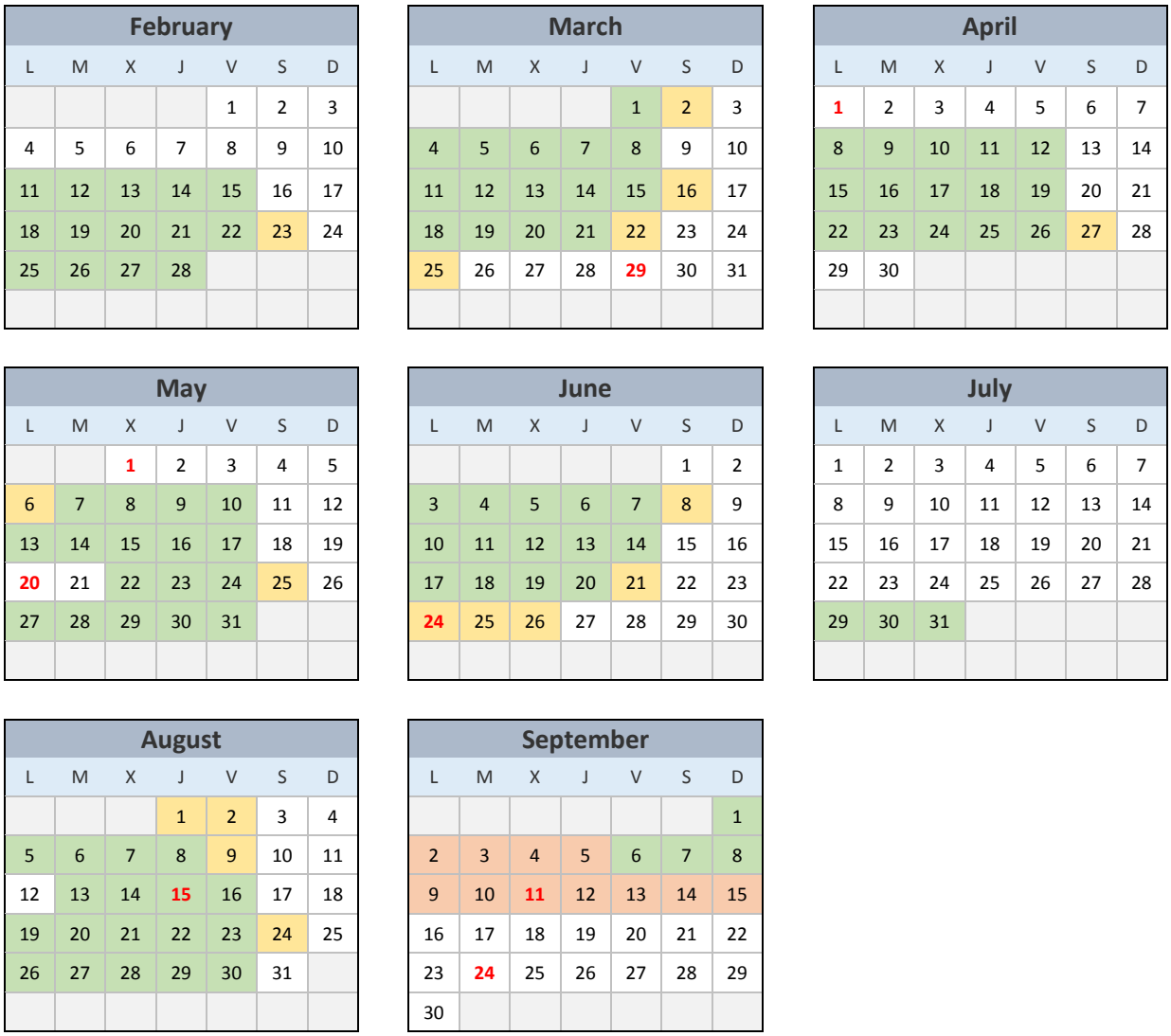


Figure 8.3. Project execution calendar

The total estimated number of hours dedicated to the project execution divided in the different tasks and compared to the estimated time of the project planning is shown in Figure 8.4.

Task	Task description	Estimated time	Real time
1	State of the art about web services.	42 h	38 h
2	Analysis of the current tools of PABRE framework.	70 h	62 h
3	Study of the available environment tools to develop the project.	21 h	44 h
4	Project environment and tools learning, installation and configuration.	84 h	138 h
5	Web service design.	70 h	65 h
6	Web service implementation.	210 h	≈ 228 h

7	Test client implementation.	56 h	42 h
8	Database migration.	35 h	65 h
9	API documentation.	20 h	14 h
10	Final report elaboration	150 h	≈ 160 h
	TOTAL	**758 h**	≈ **856 h**

Figure 8.4. Project execution tasks time

9 Conclusions

9.1 Project results

From the point of view of the results of the project it could be considered that it has been successful.

The inclusion of a web service in the PABRE system will allow requirement analysts to create or adapt their own tools to manage their software accessing to a centralized always updated SRP catalogue.

Before this project, the only ways to distribute the SRP catalogue, from the software requirements expert who manages the catalogue to the requirements analyst were:

- Mandatory using Pabre-Proj and Pabre-Man tools at both sides and configuring Pabre-Proj to connect to an updated Pabre-Man database, which is very insecure and dangerous since having the credential to access the dataset allows to modify and also destroy the catalogue.
- Sending a copy of the updated Apache Derby database with the embedded schema and data of the catalogue to requirement analysts, also forcing them to use Pabre-Proj tool. This method had the big deal of a constantly outdated SRP catalogue on the requirement analyst side and a big overwork for the catalogue manager that needs constantly send a copy of the database each time a modification is performed.

Now, deploying Pabre-WS, SRP managers can offer their requirement analysts immediate and updated access to their SRP catalogue.

On the other hand, the migration of the underlying DBSM of all the PABRE system tools has been successful, providing a better support for all the system in the client infrastructure providing higher availability, reliability and performance. A very automatized and reliable method to migrate database from both systems when is necessary using DdlUtils has been provided only needing to modify the DBMSs connection details. And finally, a very easy method to configure all the tools of the

PABRE systems can be used, providing Hibernate's configuration files for both DBMSs only needing to switch them in order to modify the underlying DBMS.

Finally, regarding the test client interface, a dynamic web page that allows to navigate the different schemas of the catalogue along their classifiers and SRPs, and shows the updated version of any selected SRP has been provided in order to replace the static and outdated copy of the SRP catalogue that was available in the PABRE system web page of GESSI group.

9.2 Personal evaluation

The global personal evaluation I bring from this project is very positive. Since it has been the first project I have completely designed and developed in completely individual process having to take into account every aspect of the project and putting in practise a wide variety of knowledge learned along all my academic studies.

I has been very lucky of being able to realize this project because I was very interested in web services technologies and learning most of the tools and technologies required or selected for this project. At the beginning I did not have any knowledge about Hibernate, Maven, Derby, JSTree, log4j and DdlUtils beyond having heard some time ago some of their names. I did have a very low knowledge using Apache Tomcat, Jersey, Git, RESTful APIs, JSON format, MySQL and JQuery but very far away of the required knowledge to develop a project like this.

Thanks to the project, I have learned to solve a lot of problems related with web services and the different tools I have used, realizing how a lot of times, something requires much more time that you think because unexpected and unknown problems appear. I have improved my problem solving skills looking for the solution in the documentation, books and internet webpages in an autonomous way. In my opinion one of the most important skills of a software engineer.

I am very pleased with the collaboration and high implication grade of Carme Quer, thesis director, and Cristina Palomares, previous developer of PABRE system tools and

current PhD student, who helped me along the elaboration of this final master thesis and gave me the necessary feedback during our weekly meetings.

My final conclusion is that all the knowledge I have acquired during the development of this project is going to be very useful in my future professional career, providing me a great experience and baggage with a lot of technologies I am interested in work with.

9.3 Future work

In this project it has been developed a web service to allow accessing the SRP catalogue that is currently managed using Pabre-Man tool. This web service is completely addressed to software requirement analysts that want to make use of the SRPs of the catalogue in their projects. One of the first modifications I think should be performed in PABRE system tools, would be to adapt Pabre-Proj tool in order to make use of the Pabre-WS API to access the SRP catalogue in place of connect to an underlying database, since with the current architecture a user of Pabre-Proj, either has to work using a local copy of the database, or if he connects to the centralized and updated database he would have complete access to the database, with the security risks that this carries.

Also in a future, it should be considered to extend the features of the web service offering also SRP experts the possibility of managing the catalogue creating, editing, deleting and classifying SRPs through the RESTful API. Of course, this kind of operations will need some authentication protocol to identify users that are allowed to manage the catalogue. This would need to extend the domain of the software including entities such as users, roles, etc…

In addition, the SRP catalogue search functionality, now, is very simple, limited and, although with the current and the expected future size of the catalogue it has a good performance, it should be considered in a future, above all if the size of the catalogue increases or more advanced searches over more SRP fields are necessary, to include an underlying search engine to manage the catalogue search requests. There are some interesting technologies to take in consideration such as Apache Lucene, Hibernate

Search and Solr that could be used to improve the performance and functionalities of the search.

Finally the inclusion of this web service in the PABRE framework has brought some new problems, since a SRP that has been requested and is returned to the client is out of the scope of the software and the client can manage it in its own way. The framework can never know whether the requested SRP has been finally used in a project or not and how it has been parameterized. Also, SRPs not available in the SRP catalogue but created by the client in their own application are not notified to the service in any way. This avoids the possibility of creating statistics of SRP used in projects in order to allow SRP catalogue manager to access a very useful knowledge to make a constant improve of the catalogue. It should be considered to offer new operations to notify the web service about the use of the SRPs of the catalogue in the projects and send information about SRPs created by the requirement analysts in order to consider their inclusion in the catalogue by the SRP catalogue manager.

10 Bibliography

Albreshne, A., Fuhrer, P., & Pasquier-Dorthe, J. (2009). *Web Services Technologies: State of the Art : Definitions, Standards, Case Study.* Université de Fribourg, Department of Informatics.

Alexander, C. (1978). *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

Cardaşim, F. (2011, 11 26). *Html 5 websockets*. Retrieved 05 2, 2013, from http://www.slideshare.net/codecampiasi/codecamp-iasi26-nov-2011web-sockets

DBCopy Plugin Team. (2013, 06 26). *DBCopy Plugin for SQuirreL SQL Client*. Retrieved from http://dbcopyplugin.sourceforge.net/

DuVander, A. (2011, 5 25). *1 in 5 APIs Say "Bye XML"*. Retrieved 07 15, 2013, from Programmableweb: http://blog.programmableweb.com/2011/05/25/1-in-5-apis-say-bye-xml/

Elksteing, M. (2008). *Learn REST: A Tutorial*. Retrieved 03 24, 2013, from http://rest.elkstein.org/2008/02/what-is-rest.html

Erl, T. (2006). *Service-Oriented Architecture, Concepts, Technology, and .* Prentice Hall Indiana.

FasterXML, L. (2013, 6 22). *Jackson Release: 2.0*. Retrieved from http://wiki.fasterxml.com/JacksonRelease20

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional. Retrieved from http://martinfowler.com/eaaCatalog/dataTransferObject.html

Introducing JSON. (2013, 07 15). Retrieved from http://json.org/

Jackson team. (2013, 8 2). *Jackson Annotations*. Retrieved from Jackson Documentation hub: https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations

jasonray. (2013, 6 22). *Serializing a POJO to xml or json using JAXB*. Retrieved from https://github.com/jasonray/jersey-starterkit/wiki/Serializing-a-POJO-to-xml-or-json-using-JAXB

JerseyTeam. (n.d.). *JAX-RS Application, Resources and Sub-Resources*. Retrieved 7 5, 2013, from Jersey 2.2 User Guide: http://jersey.java.net/documentation/latest/jaxrs-resources.html

Justin. (2010, 1 27). *JSON vs XML*. Retrieved 07 15, 2013, from Technology of Content: http://blog.technologyofcontent.com/2010/01/json-vs-xml/

Locken, S. (2013, 07 29). *The What, Why and How of JSON for EDI Integration Specialists*. Retrieved 08 15, 2013, from Aurora EDI Alliance: http://www.auroraedialliance.com/blog/bid/184388/Part-1-The-What-Why-and-How-of-JSON-for-EDI-Integration-Specialists

Lubbers, P., Greco, F., & Corporation, K. (2013, 05 06). *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*.

Martens, C. (2007, 5 8). *Sun: The bulk of Java is open sourced*. Retrieved 06 12, 2013, from InfoWorld: http://www.infoworld.com/d/developer-world/sun-bulk-java-open-sourced-003

Mulloy, B. (2013, 06 11). *API Design: Ruminating Over REST*. Retrieved 07 03, 2013

Oracle. (n.d.). *Building RESTful Web Services with JAX-RS*. Retrieved 07 20, 2013, from The Java EE 6 Tutorial: http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html

Palomares, C. (2010). *Implementació del procés d'utilització de patrons de requisits.* UPCommons.

Papazoglou, M. (2008). *Web Services: Principles and Technology.* Prentic Hall.

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision. *17th International World Wide Web Conference (WWW2008)* (pp. 805-814). Beijing, China: WWW.

Singh, Munindar, & Huhns, Michael. (2005). *Service-Oriented Computing.* Service-Oriented Computing.

Sporny, M. (2013, 07 15). *Web Services: JSON vs. XML*. Retrieved from Digital Bazaar: http://digitalbazaar.com/2010/11/22/json-vs-xml/

The Apache Software Foundation. (2013, 06 23). *Introduction to the Standard Directory Layout*. Retrieved from Apache Maven Project: http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

The Eclipse Foundation. (2013, 6 24). *Maven Integration (m2eclipse)*. Retrieved from http://eclipse.org/m2e/

The GESSI group. (n.d.). *GESSI group web*. Retrieved from PABRE Catalogue: http://www.upc.edu/gessi/PABRE/Patterns.html

W3C. (n.d.). *Web Service Description Language (WSDL) 1.1*. Retrieved 03 22, 2013, from http://www.w3.org/TR/wsdl

World Wide Web Consortium. (2009). *Team Comment on the "Web Application Description Language" Submission*. Retrieved 04 2013, from http://www.w3.org/Submission/2009/03/Comment

ANNEX I. Pabre-WS API documentation

This a copy of the API documentation elaborated using Apiary that can be found on http://docs.pabrews.apiary.io/

It is a simple guide for client developers to and users of the web service in order to know the available resources, the query string parameters and the format of the responses.

10.1 Schema resources

10.1.1 GET /ws/api/schemas

List of all schemas in the catalogue.

Example of response:

```
200 (OK)
    Content-Type: application/json
[
    {
        "name":"ISO/IEC 25010 (Extended NT, NF)",
        "description":"A product quality model composed of eight characteristics
(which are further subdivided into subcharacteristics) that relate to static
properties of software and dynamic properties of the computer system. The
model is applicable to both computer systems and software products. It has
been extended with non-technical characteristics an subcharacteristics.",
        "comments":"",
        "sources":[ ],
        "uri":"http://localhost:8080/Pabre-WS/api/schemas/720897"
    },
    {
        "name":"ISO/IEC 9126-1 (Extended NT)",
        "description":"",
        "comments":"",
        "sources":[ ],
        "uri":"http://localhost:8080/Pabre-WS/api/schemas/20480001"
    }
]
```

10.1.2 GET /ws/api/schemas/{id}{?unbinded,complete}

Retrieve full information about the specified schema

id	required	long	{id of the specified schema}
unbinded	optional	boolean	If false, patterns unbinded to any schema are not returned
complete	optional	boolean	If false, only one level in the contained hierarchy of the schema is returned

Example of response:

```
200 (OK)
    Content-Type: application/json

{
   "name":"ISO/IEC 9126-1 (Extended NT, NF)",
   "description":"",
   "comments":"",
   "sources":[ ],
   "uri":"http://localhost:8080/Pabre-WS/api/schemas/98330",
   "rootClassifiers":[
      {
         "name":"Non Functional",
         "description":"",
         "comments":"",
         "sources":[ ],
         "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98331",
         "type":2,
         "internalClassifiers":[
            {
               "name":"Functionality",
               "description":"",
               "comments":"",
               "sources":[ ],
               "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98378",
               "type":2,
               "internalClassifiers":[
                  {
                     "name":"Interoperability",
                     "description":"",
                     "comments":"",
                     "sources":[ ],
                     "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98379",
                     "type":1,
                     "internalClassifiers":[ ],
                     "npatterns":2,
                     "pos":0,
                     "requirementPatterns":[
                        {
                           "name":"Data Exchange",
                           "uri":"http://localhost:8080/Pabre-
WS/api/patterns/655416",
                           "editable":true,
                           "available":true
                        },
                        {
                           "name":"Interoperability with External Systems
",
                           "uri":"http://localhost:8080/Pabre-
WS/api/patterns/1900586",
                           "editable":true,
                           "available":true
                        }
                     ]
                  }
               ]
            }
         ]
      },
      {
         "name":"Non Technical",
         "description":"",
```

```
            "comments":"",
            "sources":[ ],
            "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98332",
            "type":2,
            "internalClassifiers":[
                {
                    "name":"Supplier",
                    "description":"",
                    "comments":"",
                    "sources":[ ],
                    "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98333",
                    "type":2,
                    "internalClassifiers":[
                        {
                            "name":"Organizational Structure",
                            "description":"",
                            "comments":"",
                            "sources":[ ],
                            "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98336",
                            "type":1,
                            "internalClassifiers":[ ],
                            "npatterns":3,
                            "pos":0,
                            "requirementPatterns":[
                                {
                                    "name":"Supplier Administrative Information",
                                    "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283546",
                                    "editable":true,
                                    "available":true
                                },
                                {
                                    "name":"Supplier Organization",
                                    "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283548",
                                    "editable":true,
                                    "available":true
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ],
    "unbindedPatterns":[
        {
            "name":"Documentation (OLD)",
            "uri":"http://localhost:8080/Pabre-WS/api/patterns/655425",
            "editable":false,
            "available":true
        },
        {
            "name":"Crash Response (OLD)",
            "uri":"http://localhost:8080/Pabre-WS/api/patterns/2031619",
            "editable":false,
            "available":true
        }
    ]
}
```

10.2 Classifier resources

10.2.1 GET /ws/api/schemas/{id}/classifiers{?complete}

List of classifiers of the specified schema

id	required	long	{id of the specified schema}
complete	optional	boolean	If false, only one level in the contained hierarchy of the schema is returned

Example of response:

```
200 (OK)
    Content-Type: application/json

[
  {
     "name":"Non Functional",
     "description":"",
     "comments":"",
     "sources":[ ],
     "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98331",
     "type":2,
     "internalClassifiers":[
        {
           "name":"Functionality",
           "description":"",
           "comments":"",
           "sources":[ ],
           "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98378",
           "type":2,
           "internalClassifiers":[
              {
                 "name":"Interoperability",
                 "description":"",
                 "comments":"",
                 "sources":[ ],
                 "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98379",
                 "type":1,
                 "internalClassifiers":[ ],
                 "npatterns":2,
                 "pos":0,
                 "requirementPatterns":[
                    {
                       "name":"Data Exchange",
                       "uri":"http://localhost:8080/Pabre-
WS/api/patterns/655416",
                       "editable":true,
                       "available":true
                    },
                    {
                       "name":"Interoperability with External Systems ",
                       "uri":"http://localhost:8080/Pabre-
WS/api/patterns/1900586",
                       "editable":true,
                       "available":true
                    }
```

```
                    ]
                }
            ]
        }
    ]
  },
  {
      "name":"Non Technical",
      "description":"",
      "comments":"",
      "sources":[ ],
      "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98332",
      "type":2,
      "internalClassifiers":[
          {
              "name":"Supplier",
              "description":"",
              "comments":"",
              "sources":[ ],
              "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98333",
              "type":2,
              "internalClassifiers":[
                  {
                      "name":"Organizational Structure",
                      "description":"",
                      "comments":"",
                      "sources":[ ],
                      "uri":"http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98336",
                      "type":1,
                      "internalClassifiers":[ ],
                      "npatterns":3,
                      "pos":0,
                      "requirementPatterns":[
                          {
                              "name":"Supplier Administrative Information",
                              "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283546",
                              "editable":true,
                              "available":true
                          },
                          {
                              "name":"Supplier Organization",
                              "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283548",
                              "editable":true,
                              "available":true
                          }
                      ]
                  }
              ]
          }
      ]
  }
]
```


10.2.2 GET /ws/api/schemas/{id}/classifiers/{id}

Retrieve full information about a specified classifier in a schema.

schemaid	required	long	{id of the schema}
id	required	long	{id of the specified classifier}

complete	optional	boolean	If false, only one level in the contained hierarchy of the classifier is returned

Example of response:

```
200 (OK)
    Content-Type: application/json

{
   "name": "Functionality",
   "description": "",
   "comments": "",
   "sources": [],
   "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98378",
   "type": 2,
   "internalClassifiers": [
      {
         "name": "Interoperability",
         "description": "",
         "comments": "",
         "sources": [],
         "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98379",
         "type": 1,
         "internalClassifiers": [],
         "npatterns": 2,
         "pos": 0,
         "requirementPatterns": [
            {
               "name": "Data Exchange",
               "uri": "http://localhost:8080/Pabre-WS/api/patterns/655416",
               "editable": true,
               "available": true
            },
            {
               "name": "Interoperability with External Systems ",
               "uri": "http://localhost:8080/Pabre-WS/api/patterns/1900586",
               "editable": true,
               "available": true
            }
         ]
      },
      {
         "name": "Accuracy",
         "description": "",
         "comments": "",
         "sources": [],
         "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98380",
         "type": 1,
         "internalClassifiers": [],
         "npatterns": 1,
         "pos": 1,
         "requirementPatterns": [
            {
               "name": "Data Precision",
               "uri": "http://localhost:8080/Pabre-WS/api/patterns/1835031",
               "editable": true,
               "available": true
            }
         ]
      },
      {
```

```
            "name": "Security",
            "description": "",
            "comments": "",
            "sources": [],
            "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98381",
            "type": 1,
            "internalClassifiers": [],
            "npatterns": 5,
            "pos": 2,
            "requirementPatterns": [
                {
                    "name": "Authentication",
                    "uri": "http://localhost:8080/Pabre-WS/api/patterns/2031814",
                    "editable": true,
                    "available": true
                },
                {
                    "name": "Authorization",
                    "uri": "http://localhost:8080/Pabre-WS/api/patterns/2031825",
                    "editable": true,
                    "available": true
                },
                {
                    "name": "Automatic Logoff",
                    "uri": "http://localhost:8080/Pabre-WS/api/patterns/2031845",
                    "editable": true,
                    "available": true
                },
                {
                    "name": "Data Transmission Protection ",
                    "uri": "http://localhost:8080/Pabre-WS/api/patterns/2031849",
                    "editable": true,
                    "available": true
                },
                {
                    "name": "Stored Data Protection",
                    "uri": "http://localhost:8080/Pabre-WS/api/patterns/2031860",
                    "editable": true,
                    "available": true
                }
            ]
        },
        {
            "name": "Suitability",
            "description": "",
            "comments": "",
            "sources": [],
            "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98382",
            "type": 3,
            "internalClassifiers": [],
            "npatterns": 0,
            "pos": 3,
            "requirementPatterns": []
        },
        {
            "name": "F. Compliance",
            "description": "",
            "comments": "",
            "sources": [],
            "uri": "http://localhost:8080/Pabre-
WS/api/schemas/98330/classifiers/98383",
            "type": 3,
            "internalClassifiers": [],
            "npatterns": 0,
            "pos": 4,
            "requirementPatterns": []
```

```
        }
    ],
    "npatterns": 0,
    "pos": 0,
    "requirementPatterns": []
}
```

10.3 Pattern resources

10.3.1 GET /ws/api/patterns?{keyword}

List of patterns in the system ordered alphabetically.

keyword	optional	string	Search string to filter and return only patterns that content this string in any of their related keywords

Example of response:

```
200 (OK)
    Content-Type: application/json

[
    {
        "name":"Acceptance Tests",
        "uri":"http://localhost:8080/Pabre-WS/api/patterns/20283601",
        "editable":true,
        "available":true
    },
    {
        "name":"Access to Customer Premises",
        "uri":"http://localhost:8080/Pabre-WS/api/patterns/20283587",
        "editable":true,
        "available":true
    },
    {
        "name":"Alternative Data Storage",
        "uri":"http://localhost:8080/Pabre-WS/api/patterns/2031738",
        "editable":true,
        "available":true
    }
]
```

10.3.2 GET /ws/api/patterns/{id}

Retrieve full information about the specified pattern

id	required	long	{id of the specified classifier}

Example of response:

```
200 (OK)
    Content-Type: application/json

{
   "name":"Interface Language",
   "description":"This pattern expresses the need of having the user interface
in some given languages",
```

```
   "comments":"",
   "sources":[
      {
         "uri":"http://localhost:8080/Pabre-WS/api/sources/786434",
         "identifier":"Requirement books from CITI"
      },
      {
         "uri":"http://localhost:8080/Pabre-WS/api/sources/786435",
         "identifier":"Specialized literature"
      }
   ],
   "author":"GESSI-CITI",
   "available":true,
   "dependencies":[ ],
   "editable":true,
   "forms":[
      {
         "name":"Homogeneous Interface Language",
         "description":"All the user interface is written in the same
languages ",
         "comments":"",
         "sources":[
            {
               "uri":"http://localhost:8080/Pabre-WS/api/sources/786434",
               "identifier":"Requirement books from CITI"
            }
         ],
         "author":"GESSI, CITI",
         "modificationDate":"2009-06-04T22:00:00.000+0000",
         "numInstances":0,
         "available":true,
         "statsNumInstances":0,
         "statsNumAssociates":0,
         "fixedPart":{
            "formText":"The system shall provide the user interface available
in %intLanguages% languages",
            "questionText":"Can yoursystem provide an user interface in
%intLanguages% languages?",
            "numInstances":0,
            "available":true,
            "statsNumInstances":0,
            "parameters":[
               {
                  "name":"intLanguages",
                  "correctnessCondition":"",
                  "description":"is a non-empty set of natural languages",
                  "metric":{
                     "name":"NaturalLanguages",
                     "description":"NaturalLanguages = Set(NaturalLanguage)
",
                     "comments":"",
                     "sources":[ ],
                     "uri":"http://localhost:8080/Pabre-
WS/api/metrics/2031900",
                     "simple":{
                        "name":"NaturalLanguage",
                        "description":"NaturalLanguage= Domain(ISO language
definition)",
                        "comments":"",
                        "sources":[ ],
                        "uri":"http://localhost:8080/Pabre-
WS/api/metrics/2031899",
                        "possibleValues":[
                           "ISO Language",
                           "..."
                        ],
                        "defaultValue":null
                     }
```

```
                    }
                }
            ]
        },
        "extendedParts":[ ]
    }
],
"goal":"Make the user interface understandable (from a language point of
view) for its users ",
"keywords":[
    "Idioms",
    "Language"
],
"numInstances":0,
"statsNumAssociates":0,
"statsNumInstances":0,
"uri":"http://localhost:8080/Pabre-WS/api/patterns/655398",
"versionDate":"2009-03-19T23:00:00.000+0000",
"versions":[
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/655398/versions/655398",
        "versionDate":"2009-03-19T23:00:00.000+0000"
    }
],
"versionUri":"http://localhost:8080/Pabre-
WS/api/patterns/655398/versions/655398"
}
```


10.4 Version resources

10.4.1 GET /ws/api/patterns/{patternId}/versions

List of versions of a pattern

patternId	required	long	{id of the specified pattern}

Example of response:


```
200 (OK)
    Content-Type: application/json

[
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283601/versions/20414488",
        "versionDate":"2012-05-23T12:31:48.000+0000"
    },
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283601/versions/393216",
        "versionDate":"2012-06-28T11:25:19.000+0000"
    },
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283601/versions/20283601",
        "versionDate":"2012-06-28T11:30:26.000+0000"
    },
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283601/versions/1081345",
```

```
        "versionDate":"2013-06-03T10:42:55.000+0000",
        "reason":"testing"
    },
    {
        "uri":"http://localhost:8080/Pabre-
WS/api/patterns/20283601/versions/1212416",
        "versionDate":"2013-06-05T10:42:24.000+0000",
        "reason":"There were errors in the database data"
    }
]
```


10.4.2 GET /ws/api/patterns/{patternId}/versions/{versionId}

Retrieve full information about the specified version of a pattern

patternId	required	long	{id of the specified pattern}
versionId	required	long	{id of the specified version of the pattern}

Example of response:


```
200 (OK)
    Content-Type: application/json

{
    "uri":"http://localhost:8080/Pabre-
WS/api/patterns/1900586/versions/1900586",
    "versionDate":"2009-03-19T23:00:00.000+0000",
    "author":"GESSI-CITI",
    "goal":"Communicate the system with other, external systems.",
    "numInstances":-4,
    "available":true,
    "statsNumInstances":0,
    "statsNumAssociates":0,
    "forms":[
        {
            "name":"Interoperability Functionality",
            "description":"This form expresses the general functionality of
interoperability with external systems and provides extensions to declare the
concrete systems and technological details. As redacted, it may be used to
state interoperability with the current system too",
            "comments":"",
            "sources":[
                {
                    "uri":"http://localhost:8080/Pabre-WS/api/sources/786435",
                    "identifier":"Specialized literature"
                },
                {
                    "uri":"http://localhost:8080/Pabre-WS/api/sources/786434",
                    "identifier":"Requirement books from CITI"
                }
            ],
            "author":"GESSI-CITI",
            "modificationDate":"2009-03-19T23:00:00.000+0000",
            "numInstances":0,
            "available":true,
            "statsNumInstances":0,
            "statsNumAssociates":0,
            "fixedPart":{
                "formText":"The system shall be able to interoperate with
external systems",
                "questionText":"Is your system able to interoperate with external
systems?",
```

```
            "numInstances":0,
            "available":true,
            "statsNumInstances":0,
            "parameters":[ ]
         },
         "extendedParts":[
            {
               "formText":"The system shall provide Application Program
Interfaces (APIs) to be used with the following technologies: %procComm%",
               "questionText":"Can your system provide Application Program
Interfaces (APIs) to be used with %procComm% technologies?",
               "numInstances":0,
               "available":true,
               "statsNumInstances":0,
               "parameters":[
                  {
                     "name":"procComm",
                     "correctnessCondition":"",
                     "description":"is a non-empty set of inter-process
communication protocols",
                     "metric":{
                        "name":"ProcessComunicationsProtocols",
                        "description":"ProcessComunicationsProtocols =
Set(ProcessComunicationsProtocol) \\n",
                        "comments":"",
                        "sources":[ ],
                        "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900593",
                        "simple":{
                           "name":"ProcessComunicationsProtocol",
                           "description":"ProcessComunicationsProtocol
=\\nDomain(SOAP, .NET, IIOP, RMI, CORBA, SQL, ODBC, â€¦)\\n",
                           "comments":"",
                           "sources":[ ],
                           "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900592",
                           "possibleValues":[
                              "ODBC",
                              ".NET",
                              "IIOP",
                              "CORBA",
                              "SQL",
                              "SOAP",
                              "...",
                              "RMI"
                           ],
                           "defaultValue":null
                        }
                     }
                  }
               ],
               "name":"E1- APIs Provided"
            },
            {
               "formText":"The system shall be able to interoperate with
external systems for performing the tasks %systFunctionalities%",
               "questionText":"Is your system able to interoperate with
external systems for performing the %systFunctionalities% tasks?",
               "numInstances":0,
               "available":true,
               "statsNumInstances":0,
               "parameters":[
                  {
                     "name":"systFunctionalities",
                     "correctnessCondition":"",
                     "description":"is a non-empty set of system
functionalities",
                     "metric":{
```

```
                            "name":"SystemFunctionalities",
                            "description":"SystemFunctionalities =
Set(SystemFunctionality)",
                            "comments":"",
                            "sources":[ ],
                            "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900591",
                            "simple":{
                                "name":"SystemFunctionality",
                                "description":"SystemFunctionality = String",
                                "comments":"",
                                "sources":[ ],
                                "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900590",
                                "defaultValue":null
                            }
                        }
                    }
                ],
                "name":"E2- Functionalities Provided Thanks Interoperation to
Other Systems"
            },
            {
                "formText":"The system shall be able to interoperate with the
%extSystems% systems for performing the tasks %systFunctionalities%",
                "questionText":"Is your system able to interoperate with the
%extSystems% systems for performing the %systFunctionalities% tasks?",
                "numInstances":0,
                "available":true,
                "statsNumInstances":0,
                "parameters":[
                    {
                        "name":"extSystems",
                        "correctnessCondition":"",
                        "description":"is a non-empty set of software names
(systems, database, etc.; it must be a subset of the first parameter in the
extension External System to Interoperate",
                        "metric":{
                            "name":"SoftwareNames",
                            "description":"SoftwareNames = Set(SoftwareName)",
                            "comments":"",
                            "sources":[ ],
                            "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900589",
                            "simple":{
                                "name":"SoftwareName",
                                "description":"SoftwareName = String",
                                "comments":"",
                                "sources":[ ],
                                "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900588",
                                "defaultValue":null
                            }
                        }
                    },
                    {
                        "name":"systFunctionalities",
                        "correctnessCondition":"",
                        "description":"is a non-empty set of system
functionalities",
                        "metric":{
                            "name":"SystemFunctionalities",
                            "description":"SystemFunctionalities =
Set(SystemFunctionality)",
                            "comments":"",
                            "sources":[ ],
                            "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900591",
```

```
                            "simple":{
                                "name":"SystemFunctionality",
                                "description":"SystemFunctionality = String",
                                "comments":"",
                                "sources":[ ],
                                "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900590",
                                "defaultValue":null
                            }
                        }
                    }
                ],
                "name":"E3- Functionalities Provided Thanks Interoperation to
Specific System"
            },
            {
                "formText":"The system shall be able to interoperate with the
%extSystems% systems",
                "questionText":"Is your system able to interoperate with
%extSystems% systems?",
                "numInstances":0,
                "available":true,
                "statsNumInstances":0,
                "parameters":[
                    {
                        "name":"extSystems",
                        "correctnessCondition":"",
                        "description":"is a non-empty set of software names
(systems, database, etc.)",
                        "metric":{
                            "name":"SoftwareNames",
                            "description":"SoftwareNames = Set(SoftwareName)",
                            "comments":"",
                            "sources":[ ],
                            "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900589",
                            "simple":{
                                "name":"SoftwareName",
                                "description":"SoftwareName = String",
                                "comments":"",
                                "sources":[ ],
                                "uri":"http://localhost:8080/Pabre-
WS/api/metrics/1900588",
                                "defaultValue":null
                            }
                        }
                    }
                ],
                "name":"E4- External System Interoperation Capability"
            }
        ]
    }
],
"keywords":[ ],
"dependencies":[ ],
"requirementPattern":{
    "name":"Interoperability with External Systems ",
    "available":true,
    "editable":true,
    "uri":"http://localhost:8080/Pabre-WS/api/patterns/1900586"
}
}
```

10.5 Metric resources

10.5.1 GET /ws/api/metrics?{complete}

List all metrics in the catalogue

complete	optional	boolean	If true, additional information about metrics contained in Sets ("simple" field) and Domain ("possibleValues" field) is returned

Example of response:

```
200 (OK)
    Content-Type: application/json

[
    {
        "name": "UptimeInstant",
        "description": "UptimeInstant = TimePoint",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/2031754"
    },
    {
        "name": "timePoint2",
        "description": "",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/17170433"
    },
    {
        "name": "Date",
        "description": "Date = TimePoint",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283450"
    },
    {
        "name": "AcceptanceCondition",
        "description": "AcceptanceCondition = String (eg. \"the implemented
system does not have a large amount of non-blocking defects\", ?defects
detected during the tests are corrected?)",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283471"
    },
    {
        "name": "AcceptanceConditions",
        "description": "AcceptanceConditions = Set(AcceptanceCondition)",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283529",
        "simple": {
            "name": "AcceptanceCondition",
            "description": "AcceptanceCondition = String (eg. \"the
implemented system does not have a large amount of non-blocking
defects\", ?defects detected during the tests are corrected?)",
            "comments": "",
            "sources": [],
            "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283471",
            "defaultValue":null
        }
```

```
    },
    {
        "name": "AcceptanceTestsStartTimePoint",
        "description": "AcceptanceTestsStartTimePoint = String (e.g. \"the date
of notification of final acceptance\")",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283469"
    },
    {
        "name": "AccessType",
        "description": "AccessType = Domain(All, Modification, Read, â€¦)",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/2031765",
        "possibleValues": [
            "All",
            "...",
            "Read",
            "Modification"
        ],
        "defaultValue": null
    }
]
```


10.5.2 GET /ws/api/metrics?{id}

Retrieve full information about the specified metric

id	required	long	{id of the specified metric}

Example of response:


```
200 (OK)
    Content-Type: application/json

{
    "name": "AcceptanceConditions",
    "description": "AcceptanceConditions = Set(AcceptanceCondition)",
    "comments": "",
    "sources": [],
    "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283529",
    "simple": {
        "name": "AcceptanceCondition",
        "description": "AcceptanceCondition = String (eg. \"the implemented
system does not have a large amount of non-blocking defects\", ?defects
detected during the tests are corrected?)",
        "comments": "",
        "sources": [],
        "uri": "http://localhost:8080/Pabre-WS/api/metrics/20283471",
        "defaultValue":null
    }
}
```


10.6 Source resources

10.6.1 GET /ws/api/sources

List all sources in the catalogue

Example of response:

```
200 (OK)
    Content-Type: application/json

[
   {
      "uri": "http://localhost:8080/Pabre-WS/api/sources/786433",
      "identifier": "ISO/IEC 9126-1",
      "reference": "",
      "type": "",
      "comments": ""
   },
   {
      "uri": "http://localhost:8080/Pabre-WS/api/sources/786434",
      "identifier": "Requirement books from CITI",
      "reference": "",
      "type": "",
      "comments": ""
   },
   {
      "uri": "http://localhost:8080/Pabre-WS/api/sources/786435",
      "identifier": "Specialized literature",
      "reference": "",
      "type": "",
      "comments": ""
   }
]
```

10.6.2 GET /ws/api/sources/{id}

Retrieve full information about the specified source

id	required	long	{id of the specified source}

Example of response:

```
200 (OK)
    Content-Type: application/json

{
   "uri": "http://localhost:8080/Pabre-WS/api/sources/786433",
   "identifier": "ISO/IEC 9126-1",
   "reference": "",
   "type": "",
   "comments": ""
}
```