Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. L. Martens
i.s.m. Laboratoire Informatique Grenoble (LIG) - équipe SIGMA

Maintenance corrective et maintenance évolutive
de l’application AGAP
(Maintenance of the AGAP application by correcting and upgrading)

door

Wouter Debie

Promotoren: Prof. Dr. Ir. L. Martens (Ugent), Agnès FRONT (ENSIMAG)
Scriptiebegeleiders: Nicolas Arnaud (LIG-SIGMA)

Scriptie ingediend tot het behalen van de academische graad van
Licentiaat Informatica
optie toegepaste informatica

Academiejaar 2006–2007
Word of gratitude

First I would like to thank Mr Jean-Pierre GIRAUDIN who had confidence in me and integrated me in his research team SIGMA.

Thanks to Mr Nicolas ARNAUD for his patience, help, explanations, enthusiasm, the way he helped me throughout my internship. Mrs. Agnes FRONT who helped also in with my work.

Also I would like to thank the SIGMA team for the super atmosphere that allowed me to perform my internship in the best conditions.

Thanks to ENSIMAG (École Nationale Superièure d’Informatique et Mathématique Appliquer Grenoble) who accepted me as an Erasmus student here in France and gave me the opportunity to follow this internship.
Toelating tot bruikleen

“De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.
Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.”

“The author gives the permission to consult this thesis and to copy parts of it for personal use.
Any other use falls under the restrictions of the copyright, in particular concerning the obligation to mention the source explicitly when quoting results from this thesis.”

Wouter Debie, juni 2007
Maintenance of the AGAP application by correcting and upgrading

by

Wouter Debie

Project for the 3rd grade ENSIMAG

year 2006-2007

ENSIMAG

Tutor: Agnes FRONT

training period responsible: Nicolas ARNAUD

Summary

The purpose of this training was to assure the maintenance of AGAP, a development environment for defining and using patterns. Its aim is to help the patterns engineer and the application engineer. The first is able to define several patterns system using the same formalism or even items of existing formalisms. The application engineer specifies his own information systems by reusing patterns.

First, this paper deals with AGAP’s context, then with technical aspects and technologies followed by the maintenance operations we’ve preformed on AGAP (DAO, iBATIS). The last but biggest part talks about the analysis and design of a rights management policy which we’ve introduced to create a rights policy on formalisms, patterns and pattern systems. This system provides protection on the work of every user and adds possibilities for working in teams.

Proposed by: Laboratoire Informatique Grenoble (LIG) équipe SIGMA

Mots-cle: patron, système de patrons, atelier à base de patrons, gestion des droits.

Key-words: patterns, patterns system, pattern-based tool, rights management policy.
Maintenance corrective et maintenance évolutive de l’application AGAP
(Maintenance of the AGAP application by correcting and upgrading)

Wouter Debie
Supervisor(s): Agnès FRONT, Nicolas ARNAUD

Abstract—This article tries to add a rights management policy to the AGAP (Atelier de Gestion et d’Application de Patrons) system which is a tool that manages formalisms, pattern systems and information systems. AGAP is used by 2 types of engineers: pattern engineers, who create pattern systems, and application engineers, who use these patterns in the design of applications. This rights management policy has to protect the work of each of these users.

Keywords—patterns, patterns system, pattern-based tool, rights management policy.

I. INTRODUCTION

In the current version of AGAP (Atelier de Gestion et d’Application de Patrons) there are no access rights on the objects created by the users. Every user can change the work of an other user. It is better to add access right to the system. Only the owner may modify or delete his work.

II. THE AGAP APPLICATION

The AGAP application is created by the SIGMA (Système d'Information - inGénierie et Modélisation Adaptables) research-team whom is part of the LIG (Laboratoire Informaticque Grenoble). This is application where you can manage formalism, pattern systems and information systems.

A. users

In AGAP we have 3 types of actors who interact with the application. First of all a pattern engineer. He can create a new formalism and use them during the development of a pattern and/or pattern systems. The second type of user is the application engineer who uses the pattern systems and patterns to create an information system. The last type is the administrator. This user adds, deletes users and performs other administration tasks.

B. Technology

The AGAP application is a web application based on a 3-tier architecture. At the application-tier we find an APACHE/TOMCAT web server and at the database-tier a POSTGRESQL server. The interaction with the users is coded in jsp and java files whom are compiled by the TOMCAT server.

III. THE RIGHTS MANAGEMENT SYSTEM

A. Problem

At this point of the AGAP project there aren’t any access rights installed. Every user is capable to modify or even to delete the work of an other user. The problem becomes greater when AGAP is going to be shared between several different research teams. It’s not possible to hide patterns from other teams.

B. Requirements Analysis

To add security to the AGAP application we need the following things for the 3 components: formalism, pattern system and pattern:

- Formalism
  - An owner (patterns Engineer) who can modify it,
  - Can be seen by everybody after validation,
  - Once it is validated it can’t be modified anymore.

- Pattern System
  - A supervisor (or person in charge) whose functions are:
    * Adding/Deleting a contributor
    * Adding a supervisor (or person in charge)
    * supervisor ⇒ contributor
    * Validating for contribution ⇒ the formalism can’t be changed anymore
    * Publication
    * Modifying the header
    * Check (or mark) pattern for deletion
  - Has several contributors who add patterns to the system

- Patterns
  - A contributor whose functions are:
    * Modifying the pattern
    * Deleting the pattern
    * Validating the pattern (After this, only enriching the pattern is possible)

C. Userspace and Systemspace

Userspace This is the private space for each user. Here he can create objects in a private box only accessible for him. An other term is “sandbox”.

Systemspace This is the global space visible for every user. For each object in this space, each user has rights for the visualisation/modifications/deletion of it.

With the function Export we can transfer an object from the userspace to the systemspace. Not every user will be allowed to perform this action.

D. Formalism

A pattern engineer can create a formalism in his sandbox. He will become the owner of this formalism. There can only be one owner for one formalism. This owner may modify or delete it. Once it has been validated it can’t be modified.

Only a formalism engineer may publish a formalism after val-
idation. Once a formalism is publish it’s located in the systemspace and visible for every pattern engineer.

E. pattern system

A pattern engineer can also create a system of patterns. This user becomes the supervisor of this pattern system. This system is located in the userspace of the supervisor. This person may

• Modify the header of the system
• Validate the header for contribution
• Add users to help contribute (must be exported)
• Delete the system

A pattern engineer has the capability to export his systems of patterns for contribution and may publish it. Publishing a system of patterns makes an end to the contribution to this system.

F. pattern

A pattern engineer who creates a pattern becomes the contributor of this pattern. This gives him the right to modify, delete, validate and enrich this pattern. Notice that you have to be added as contributor to the system of patterns to be able to create a pattern for this system. Another remark is that the supervisor of the pattern system can enrich every pattern in its system. On the other hand, a contributor to a system of patterns may only enrich the patterns he created.

G. Groups

Now we’ve developed an access rights system for the creation and modification of objects, we can wonder what happens after publication. Because AGAP will be used by several different companies, it’s better to add here also access rights. Because the systems of patterns are created by a team (the supervisor and the contributors) they have to be able to visualize their creation even after they have been published. To solve this problem we need the notion of groups. We distinguish two types of groups.

Real group A real group represents a real social group like a group of co-workers, an enterprise, ... This type of group contains other real groups and individuals. The groups are created by the administrator of the AGAP application.

Group of contributors of a system of patterns Every system of patterns has a group of contributors. They contain real groups or individuals. It is the responsibility of the supervisor to control this group.

Group of visitors of a system of patterns When we publish a system of patterns we create automatically a new group: group of visitors. This group contains Real Groups and individuals who have the permission to visualize this system. It’s the task of the supervisor to fill this group up at the moment of the publication. Note that the group of contributors of this system of patterns is automatically included in the group of visitors. The persons who helped to create this system have of course the right to visualize the system after publication.

IV. Conclusion

The AGAP system has now a policy for visualizing and modifying objects. Now, every user has his objects and he only is able to modify them. It contains also the notion of group which brings it closer to the reality, where users work in teams. During the changes we’ve made to the AGAP project we’ve also cleaned up the code and installed the iBATIS framework for better Control of the I/O to the database.

REFERENCES

[1] Laurent TASTET AGAP: un Atelier de Gestion et d’Application de Patrons, Conservatoire national des arts et metiers Centre Regional Rhône-Alpes Centre d’enseignement de Grenoble, 2004

[2] Emmanuel Jaussier Démarche Symphony étendue, formalisation et expérimentation sur un Système d’Information Hospitalier, Conservatoire national des arts et metiers Centre Regional Rhône-Alpes Centre d’enseignement de Grenoble, 2005

# Contents

1 The project ................................. 1

2 Patterns .................................... 3
   2.1 History ................................. 3
   2.2 Definition of a pattern ............ 3
   2.3 Classification of patterns ....... 3
   2.4 Use .................................. 4
   2.5 Example .............................. 4
   2.6 System of patterns ................. 6
   2.7 Formalism ............................ 7

3 AGAP ........................................ 8
   3.1 Users .................................. 8
      3.1.1 Pattern engineer ................ 8
      3.1.2 Application engineer .......... 8
      3.1.3 Administrator .................. 8
   3.2 Components of AGAP ............... 9
   3.3 Functionalities of AGAP .......... 10
   3.4 Technologies ......................... 10

4 Correcting and Updating AGAP ........ 12
   4.1 Use of a Data Mapper Framework .. 12
      4.1.1 Introduction .................... 12
      4.1.2 iBATIS .......................... 12
      4.1.3 iBATIS life cycle ............. 12
      4.1.4 Installation .................... 13
      4.1.5 Programming with Data Mapper 14
   4.2 Using a Data Access Object ........ 16
      4.2.1 DAO Pattern .................... 16
      4.2.2 DAO in AGAP ................... 17

5 Introducing a right management policy 19
   5.1 Problem ................................ 19
   5.2 Requirements analysis ............. 19
List of Figures

2.1 UML representation of a pattern ........................................ 4
2.2 Structure of the MVC pattern ............................................. 6
3.1 AGAP users ......................................................................... 9
3.2 Components of AGAP .......................................................... 9
3.3 Screenshot AGAP: Creating a formalism ............................... 10
3.4 3 tiers architecture ............................................................. 11
4.1 iBATIS ................................................................................. 13
4.2 iBATIS flow ....................................................................... 14
4.3 Data Access Object .............................................................. 16
4.4 The DAO model in AGAP ..................................................... 17
5.1 Use case for formalism component ....................................... 21
5.2 Use case for System of patterns component ....................... 23
5.3 Use case for supervisor ....................................................... 23
5.4 Use case for contributor ..................................................... 24
5.5 users in AGAP ................................................................. 25
5.6 Creation of a user by users with administrator-rights .......... 26
5.7 Publishing a pattern system ............................................... 27
5.8 The group component ........................................................ 29
5.9 Screenshot of interface for creating a group ....................... 30
5.10 Sandbox .......................................................................... 31
5.11 Struts workflow ............................................................... 32
5.12 Screenshot of the error handling in AGAP ......................... 33
5.13 The four components of AGAP expressed in the Symphony formalism ........................................ 35
Chapter 1

The project

An internship at ENSIMAG (Ecole Nationale Superi` eure d’Informatique et Math´ ematique Appliquer Grenoble) is a way to complete our formation of engineer. You get in touch with the industry and are able to put your knowledge into practice. I followed my internship in the laboratory LIG (Laboratoire d’Informatique Grenoble). In order to let you situate the environment of my internship I will give a brief description of this organisation. The LIG is the new informatics laboratory of Grenoble. Until 2007 it was known under the name LSR (Logiciels, Syst` emes, R´ eaux) of the IMAG (Informatique et Math´ ematiques Appliqu´ ees de Grenoble). Its scientific goal is:

- Propose models and algorithms for the evolution of the discipline,
- Implement these in software,
- Deploying them on platforms of innovative experiments.

The LIG is financed by the CNRS (Conseil National de la Recherche Scientifique), INPG (Institut National Polytechnique de Grenoble), INRIA, UJF (Universit´ e Joseph Fourier) and UPMF (Universit´ e Pierre-Mendˆ es-France) and contains 24 teams.

The team where I carried out my internship is SIGMA (Syst` eme d’Information - inG´ enierie et Mod´ elisation Adaptables). There main activities are the design, development and exploitation of pattern libraries, formalisation of information systems, meta modeling, . . .

In SIGMA my work mainly concerned AGAP (see chapter 3 team). The AGAP application was created by the SIGMA crew of the laboratory LIG with goal to define, manage and use patterns integrated in a catalogue. My job consisted out of correcting and upgrading the application. The goal was to add a rights management policy to the system to prevent users to mess around with the work of another user.

At the beginning of my internship, I was working on the debugging of the whole application and testing it on a new version of TOMCAT and Postgresql driver. There weren’t a lot of bugs but the source code had become a bit unreadable by several modification
during the years. Also it contained a lot of unused and unneeded variables. I worked also on the improvement of the code by rewriting it to match the generic type of java 5.0.

The second part of the internship were the analysis, design and development of the management system of rights: The goal was to add a policy for visualizing, modifying and validating objects in AGAP.

The first part of this report presents a brief explanation of patterns followed by an introduction to the AGAP system. The second part contains a description of the modifications I made for improving AGAP and in the last and biggest part, I will talk about the analysis, design and development of the rights management policy.
Chapter 2

Patterns

In this first chapter I will explain what patterns are, what they represent and why they are used, what are the benefits of using patterns in a design . . .

2.1 History

The concept of patterns was introduced by the architect Christopher Alexander in 1977. He proposed a catalogue of classical problems in the construction of buildings together with there classical solutions. In 1987, Kent Beck and Ward Cunningham began experimenting with the idea to implement patterns into software development and presented there first results at the OOPSLA conference. In 1994 the book “Design Patterns: Elements of Reusable Object-Oriented Software” (Gamma) was published and in that same year the first “Pattern Languages of Programs” conference was held. One year later, the “Portlands Pattern Repository” was created.

2.2 Definition of a pattern

In software engineering a design pattern is a general repeatable solution to a commonly occurring problem in software design. It doesn’t contain source code and neither you can translate it into code. It’s a template for how to solve a problem that can be used in many different situations. A pattern proposes a solution for a problem in a certain context.

2.3 Classification of patterns

Design patterns can be classified in terms of the underlying problem they solve.

Type of knowledge A pattern capitalizes products - a product corresponds to a goal to reach - or capitalizes processes - a process corresponds to a way to reach a result.

Coverage A pattern coverage may be generic (resp. domain, enterprise) if it solves a problem frequently occurring in several domains (resp. in an application domain, in a particular enterprise).
Range The pattern range may be analysis, design or implementation depending on the stage of the engineering process it addresses.

2.4 Use

Using patterns in a design brings great advantages in terms of quality, evolution and maintenance for the software. In the production of software, patterns play different roles. The first one is to help with the validation of the specifications of the software by proposing solutions for the analysis and its conception. It permits the developers to create reusable conceptions.

It helps also to easily respect the already existing design models by verifying the use of patterns where they are necessary and evaluating its quality.

Using patterns makes the coding easier thanks to the information and instructions supplied in the documentation of patterns. In this way a good implementation is achieved.

The documentation of the program is improved by the use of common vocabulary that we find in the documentation of patterns. This way we can make reference to a higher level of abstraction.

2.5 Example

For an example of a pattern, we present the MVC Model-View-Controller (MVC) pattern.
<table>
<thead>
<tr>
<th><strong>Context</strong></th>
<th>Application presents content to users in numerous pages containing various data. Also, the engineering team responsible for designing, implementing and maintaining the application is composed of individuals with different skill sets.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
<td>Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a Java(^T)M/Swing interface for administrators, and a XML-based Web service for suppliers.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td>By applying the Model-View-Controller (MVC) pattern you separate core business model functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.</td>
</tr>
<tr>
<td><strong>Participants and Responsibilities</strong></td>
<td>The MVC architecture has its roots in Smalltalk, where it was originally applied to map the traditional input, processing, and output tasks to the graphical user interaction model. However, it is straightforward to map these concepts into the domain of mult-tier enterprise applications.</td>
</tr>
<tr>
<td><strong>Model</strong></td>
<td>The model represents enterprise data and the business rules that govern access to an update of this data.</td>
</tr>
<tr>
<td><strong>View</strong></td>
<td>The view reads the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view’s responsibility to maintain consistency in its presentation when the model changes.</td>
</tr>
<tr>
<td><strong>Controller</strong></td>
<td>The controller translates interactions with the view into actions to be performed by the model.</td>
</tr>
<tr>
<td><strong>Consequences</strong></td>
<td><strong>Re-use of Model components</strong> The separation of model and view allows multiple views to use the same enterprise model. Consequently, enterprise application’s model components are easier to implement, test, and maintain, since all access to the model goes through these components.</td>
</tr>
<tr>
<td><strong>Easier support for new types of clients</strong> To support a new type of client, you simply write a view and some controller logic and wire them into the existing enterprise application.</td>
<td></td>
</tr>
<tr>
<td><strong>Increased design complexity</strong> This pattern introduces some extra classes due to the separation of model, view and controller.</td>
<td></td>
</tr>
</tbody>
</table>
2.6 System of patterns

Whatever there range (analysis, design, implementation, ...) might be, we can always group patterns into system of pattern or catalogues. The notion “Pattern System” was also introduced by the architect C. Alexander. He gave the following definition: “A pattern system is a collection of patterns creating a vocabulary whom allows to understand and communicate ideas.”

Patterns, in a pattern system, are ordered by category and entirely connected with each other by an identical context. These systems have for aim to solve complexer problems because each pattern gives a solution for a specific problem.

A system of patterns guides the designer through the different architects. A system of patterns well designed must permit the application engineer to freedomly express his problem and create his own solution for his demands, and all of this in the context in which we can apply the pattern.

The patterns within a system of patterns are organised according to a formalism.
2.7 Formalism

A pattern consists out of 3 parts “the problem, the context and the solution” which explain the relationship between the problem and the solution within a context. A formalism is a structure used by the engineer to characterize his patterns. There are 2 types of formalisms:

- structural formalisms,
- narrative formalisms.

Structural formalisms are more interesting to represent a pattern than narrative formalism because they are more informal and little structured. A structural formalism is composed out of several columns.

A formalism is a resource to represent a pattern with purpose to explain where to use this pattern. It explains with the help of rubrics how the pattern is used and its goal. You can see a formalism like a standardized way to represent a pattern with fields like NAME present in every formalism.
Chapter 3

AGAP

AGAP (Atelier de Gestion et d’Application de Patrons) is a tool used by 2 types of engineers: pattern engineers, who create systems of patterns, and application engineers, who use these patterns in the design of applications. These 2 actors can perform several different actions within the AGAP system. There is also an administrator to handle the administration side of the AGAP system. AGAP manages formalisms, systems of patterns and information systems.

3.1 Users

3.1.1 Pattern engineer

He can create a new formalism, use it in the creation of a pattern and/or system of patterns. In a pattern system he can make links between patterns and so organise his system.

3.1.2 Application engineer

He can view systems of patterns, use patterns, integrate patterns, ... in order to create an information system.

3.1.3 Administrator

The administrator can view, add and modify users and their functions. He has also the right to delete patterns, systems of patterns, field types, domains, ... 

In figure 3.1 you have an overview of the different types of users in AGAP and the general actions they can perform.
3.2 Components of AGAP

AGAP consists out of 5 components: Users, System of patterns, Domain and technology, Field types and Formalism.
3.3 Functionalities of AGAP

In AGAP there’s a distinction between a formalism and a system of pattern. This difference lets the patterns engineer to create several different systems of patterns while using the same formalism or reusing columns from a formalism in order to make it easier for engineering patterns and increase the level of reuse.

A formalism is associated to several types of fields which correspond to types of solutions produced by the pattern. The system contains a domain and a technology for search facilities. The domain relates to the range (example: Analysis or design) while technology relate to the technology in which the systems addresses (example: object technology).

Figure 3.3 is a screenshot of AGAP. In this screenshot you see the interface of the creation of a formalism.

![Figure 3.3: Screenshot AGAP: Creating a formalism](image)

3.4 Technologies

AGAP is build on a 3-tiers architecture: “Presentation layer (client layer) , Application layer (web layer) and Data layer (database layer)”. At the Visualisation/presentation part we have HTML pages in a web browser. The client is a thin-client and doesn’t need any
extra software. The system can be viewed on any system that can display HTML-pages.

At the application level, we find a server who takes charge of the requests of the client. He retrieves the data from the database, transforms it into HTML-pages and sends them to the client. At the data side, we have a database who stores the data and who supplies the data requested by the application server.

The use of this architecture permits the use of cheap hardware to use the application. The reliability is strongly improved by the centralisation of the data. Currently in the AGAP application, we use a TOMCAT server at the application layer. It’s a server created by ASF (Apache Software Foundation). This foundation has also created the known APACHE server who is used in almost half of the web servers around the globe. At the Data layer, it has a PostgreSQL server.

The application is written in Java (1.4). Java is a portable object-oriented language so it will run on any machine in the future. For the visualisation, we use JSP (Java server pages) and several tag-libs. The Java class files, JSP-pages and tag-libs are connected to each other by the use of MVC2 (Model View Control version 2). This model is implemented thanks to the use of the Struts framework. Struts is a framework created by the ASF and is used inside the Jakarta project.
Chapter 4

Correcting and Updating AGAP

During the internship I’ve installed 2 new concepts (Data Mapper Framework and Data Access Object) in AGAP to improve the input/output of the database. The following 2 sections will explain these concepts.

4.1 Use of a Data Mapper Framework

4.1.1 Introduction

Currently in AGAP we find SQL-statements inside java-classes and JSP-files. A connection is obtained by a connection-object using the java.sql.Connection class. We can imagine the work that has to be performed in case of a change of structure in the database (adding, deleting, renaming a column/table, . . . ). To reduce the work to be performed for these modifications we are going to use iBATIS.

4.1.2 iBATIS

The iBATIS Data Mapper framework\footnote{\cite{Beg06} and \cite{iBa}} (figure 4.1) makes it easier to use a database with Java and .NET applications. iBATIS couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the iBATIS Data Mapper over object relational mapping tools. iBATIS maps JavaBeans to SQLStatements using a very simple XML descriptor. Thanks to the iBATIS Data Mapper API you can map JavaBeans objects to PreparedStatement parameters and ResultSets. It can even transform ResultSets to XML-files.

4.1.3 iBATIS life cycle

Data Mapper provides a very simple framework for using XML descriptors to map JavaBeans, Map implementations, primitive wrapper types (String, Integer,...) and even XML documents to a SQL statement. The following is a high level description of the life cycle (see figure 4.2).
1. Provides an object as a parameter (JavaBean, Map or primitive wrapper). The parameter object will be used as input values in the SQLStatements.

2. Execute the mapped statement. The Data Mapper framework will create a PreparedStatement instance, set the parameters using the provided parameter object and execute the statement.

3. In the case of an update, the number of rows effected is returned. In the case of a query, a single object, or an collection of objects is returned. Like parameters, result objects can be a JavaBean, Map, primitive type wrapper or XML.

4.1.4 Installation

Installing the iBATIS Data Mapper Framework is a matter of placing the appropriate JAR file on the classpath. In AGAP we have to place the JAR file in the /WEB-INF/lib directory. Now we still have to create the XML configuration file because the Data Mapper is configurated using a central XML configuration file containing the details of the datasource, maps and other specifications.

SQLMapConfig.xml This file configures the iBATIS Data Mapper. It gives information about the datasource (SimpleDataSourceFactory, DbcpDataSourceFactory or JndiDataSourceFactory), its transaction manager (JDBC, JTA, EXTERNAL) and cachemodel, the different sqlmaps that will be used, The TypeAlias for referring to long object-names,....
**Figure 4.2**: iBATIS flow

**SQLMAP.xml** These files contain the SQL mapped statements (*Select, Insert, Update, Delete*), specific cachemodel for statements, the parametermaps/classes (*represents input*) and resultmaps/classes (*represent output*) for the statements, type Aliases.

In appendix [A](#) and [B](#) you can find a copy of the configuration file and an example of a SQLMAP that we are going to use in the AGAP project.

### 4.1.5 Programming with Data Mapper

The SqlMapClient API has 4 primary functions:

1. Configure an SQL Map,
2. Execute an SQL update (incl. insert and delete),
3. Execute a query for a single object,
4. Execute a query for a list of objects.

**Configuration**

Once we have created the SQLMAP XML definition files and the SQL Map configuration file, we can create SqlMapClient instances by using the static method buildSqlMap() of the SqlMapClientBuilder.

```java
String resource = "url/to/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```
Transactions

There are two types of transactions: the automatic transaction and explicit transaction. The automatic transaction is executed by the framework itself. Every time we execute an update action (insert, update, delete) the framework will execute a transaction. For the explicit transaction, the SqlMapClient interface supplies 3 methods:

```java
public void startTransaction() throws SQLException
public void commitTransaction() throws SQLException
public void endTransaction() throws SQLException
```

Note that every transaction has to be ended before starting a new transaction even if you don’t commit the transaction.

Executing Statements

The SqlMapClient provides an API to execute all mapped statements associated to it. Below, we have a short list of the most used statements of this API. For every method we have 2 versions: one with a parameter Object and one without the parameter Object. The insert(), update() and delete() methods are specifically meant for update statements. You can not perform a query (exception will be thrown).

```java
public Object insert(String statementName, Object parameterObject)
    throws SQLException
public int update(String statementName, Object parameterObject)
    throws SQLException
public int delete(String statementName, Object parameterObject)
    throws SQLException
public Object queryForObject(String statementName, Object parameterObject)
    throws SQLException
public List queryForList(String statementName, Object parameterObject)
    throws SQLException
```

This framework is going to be used in the new DAO (Date Access Object) framework of AGAP.
4.2 Using a Data Access Object

In the previous version of AGAP, the object, which represents for example a formalism, was responsible for its own I/O. It was the task of this object to create a connection with the database (datasource) and store its information into it. The object was supplied with methods like toDB() and fromDB(). The problem with this solution is that if we have several different ways to store the information, we have to implement them all inside the information object (in example: formalism). And if we create a new way of storing we have to modify every information object to use this new way of storing. It is not good design if there is a strong relationship between the information and the way of storing it in a data source. Such code dependencies in components make it difficult and awkward to migrate the application from one type of data source to another.

To improve this we implemented the DAO(Data Access Object) Pattern

4.2.1 DAO Pattern

By using a Data Access Object we abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to get and store data.

In the implementation of the DAO we find the mechanism to work with the Data Source. As already mentioned the Data Source can be any type of database. The business component that relies on the DAO uses the simpler interface exposed by the DAO to its clients. The business component does not have to know the implementation of the data source. Because the interface of the DAO does not change when the underlying DAO implementation changes, this pattern allows the DAO to adapt to different storage schemes. There is no code dependency between the business component and the data source. The DAO is an adapter between the component and the data source (see figure 4.3).

![Data Access Object Diagram](dao.png)

Figure 4.3: Data Access Object
4.2.2 DAO in AGAP

The DAO model in AGAP

Inside AGAP we have several objects (formalism, pattern system, domain, ...) which must be stored into a datasource. For each of these business component we create a DAO. Each of these DAO will implement an interface. So each business component has a DAO who implements an interface created specially for this Business component.

DAOFactory

To make this design more flexible, we add a factory object. If we need in the code a DAO, we let it be created by the factory class. The factory class contains several static methods of the type getFormalismDAO(). These return the demanded DAO object ready to use.

Figure 4.4: The DAO model in AGAP

Inside the DAO Objects

For each business component in AGAP, we have created a PostgresDAO which implements the business DAO interface. For the input / output to the datasource we use the iBATIS framework that we have created. This way, the code inside the DAO becomes very simple. For example the method findUser(int userid) of the UserDAO is the following:

```java
public User findUser(int userID) throws SQLException {
    return (User) SQLMAP.queryForObject("find", userID);
}
```
where userID is the ID of the user we want to obtain and SQLMAP is a SqlMapClient object from the iBATIS framework. If we compare this with the previous method of retrieving an user in AGAP we see that it has become easier and less work for programming which implies more quality.

```java
public void Recup_user(int code){
    Connexion con = new Connexion();
    String req;
    ResultSet rs;
    con.connecter();
    req = "SELECT * FROM users WHERE usercode = "+code");
    rs = con.resultforselect(req);
    con.deconnecter();
    try{
        rs.beforeFirst();
        while(rs.next()){  
            user.setUSERCODE(code);
            user.setLOGIN(rs.getString("login");
            // and so on
        }
    }catch (SQLException e){}
}
```
Chapter 5

Introducing a right management policy

5.1 Problem

At this point in the development of the AGAP system, there was no access rights installed in the system. Every user was capable to modify and even to erase patterns created by a colleague. The goal of the second part of my work was to install a policy that only the creator of a pattern has the capability to modify and/or remove the pattern. There are still some open questions in this matter: May everybody see a pattern in development, can an author give his right temporary to an other user...

5.2 Requirements analysis

After the first brainstorm, we arrived at the following specifications.

- Formalism
  - Has an owner (patterns Engineer) whose functions are:
    * Creation
    * Modification
    * Deletion
    * Validation
  - Can be seen by everybody after validation,
  - Once it is validated it can’t be modified anymore.

- Pattern System
  - Has supervisor (or person in charge) whose functions are:
    * Adding a contributor
    * Deleting a contributor
    * Adding a supervisor (or person in charge)
• supervisor ⇒ contributor
• Validating for contribution, ⇒ the formalism can’t be changed anymore
• Publication
• Modifying the header
  • Name
  • Domain
  • Technology
  • Formalism
• Check (or mark) pattern for deletion
  – Has several contributors whose functions are:
    • Creating a pattern
    • Modifying a pattern
    • Deleting a pattern

• Patterns
  – Has a contributor whose functions are
    • Modifying the pattern
    • Deleting the pattern
    • Validating the pattern (After this, only enriching the pattern is possible)

Moreover we need the following things in AGAP: rights on the visualisation of objects, rights on the modification of objects, notion of supervisor and working in groups for the system of patterns.

5.2.1 Influence on the current concept
Creating this management of rights will have an impact on several components (formalism, user and system of patterns) and even create a new component **Group**.

5.2.2 UserSpace vs SystemSpace
To design all of this we are adding a new concept: **UserSpace** and **SystemSpace**. The UserSpace can be seen as a “sandbox”, it’s a private space for the user. Everything he makes/creates inside this space is private and only for him. On the other side we have the SystemSpace. This place is a global space accessible for every user. Inside this space we apply rights on the objects.

To transfer object from one space to the other we create the “export” function. There is a condition linked to the action: the object has to be validated and the user has to have the permission to export his objects.
5.3 Components specification

5.3.1 Formalism component

A patterns engineer can create a formalism in his sandbox. This user becomes automatically owner of this formalism. From this moment, he can express the following rights on this formalism:

- Modify (Constraint: formalism not validated)
- Delete (Constraint: formalism not validated)
- Validate (Impact: formalism becomes fixed, no modifications possible)
- Publish (Constraint: formalism has to be validated and user has to be a formalism engineer)

The following rules are obtained for creation: Only a formalism engineer may publish (export from UserSpace to SystemSpace) his formalisms. Every formalism that’s been created finds itself inside the UserSpace of the owner and can be published or exported later on to the SystemSpace. And for visualisation: a patterns engineer can visualize the formalisms which are validated an published and the formalisms in his own sandbox.

In the figure 5.1 a schematic view of the rules and functions declared for the component Formalism is shown.

Figure 5.1: Usecase for formalism component
Implementation

For the formalism component we are going to add the notion of owner. This will be done by adding a column owner to the formalism table. This column will contain the userID and will refer to the users table. Because in the old table the primary key is the name of the formalism we can not change its name once the formalism is in the database. We’re adding a new numerical identifier to the formalism table. This ID will be the new primary key and will be automatically generated by a sequence. A lot of tables refer to the primary key of the formalism so we have to adapt these foreign keys as well.

The implementation of the rights is done by adding constraints to the SQL statements. When a user logs in, we’re saving his ID in the session and use it to verify if he can modify a formalism.

5.3.2 System of patterns component

System of patterns

A pattern engineer can also create a system of patterns. This person becomes the supervisor of this system of patterns. He has the right to perform the following actions on this system of patterns:

- Modify the header of the system
- Validate the header for contribution
- Add users to help contribute
- Delete the system of patterns *(May not be validated for contribution)*

A system of patterns engineer has the capability to export his systems of patterns for contribution and may publish his systems of patterns. Publishing a system of patterns makes an end to the contribution to this system.

Pattern

A pattern engineer who creates a pattern becomes a contributor of this pattern. This gives him the right to modify, delete, validate and enrich this pattern. Notice that you have to be added as contributor to the system of patterns to be able to create a pattern for this system. An other remark is that the supervisor of the pattern system can enrich every pattern in its system. On the other hand, a contributor to a system of patterns may only enrich the patterns he created.

The supervisor of a system of patterns may mark a pattern “to be erased”. It is the responsibility of the contributor of this pattern to remove the pattern from this system. Because we don’t want to delete intellectual work, we add the function to put this pattern back in the sandbox, where it stays until the contributor finds another system where it can be used.
Figure 5.2: Usecase for System of patterns component

Figure 5.3: Usecase for supervisor
Implementation

For this component first we add a unique ID to each pattern system. This implies a lot of work in changing the database and the application because the primary key changes from the systemname to an ID. Once this work is done we can proceed by adding the notion of supervisor to the pattern system. This supervisor is represented in the database by his ID. Because a pattern system can have more than one supervisor we need to create a new table that combines the ID from the pattern system with the ID from the supervisor.

The third thing to add to this component is the notion of contributor. As explained in the analysis, each pattern system has a group containing the contributors of this pattern system.

5.3.3 User component

Each user in the AGAP system is an application engineer. Some of these users are pattern engineers. They have a sandbox where they can create the objects they want. If they create a formalism they become the owner of this formalism. In fact they play the “role” as owner of this formalism. If this owner wants to publish his formalism he has to be a formalism engineer. This type of user is a specialization of a pattern engineer.
If the pattern engineer creates a “system of patterns” in his sandbox he becomes the supervisor of this system. It’s the same as the owner of the formalism, the pattern engineer plays the “role” as supervisor for this system of patterns. If the supervisor wants to export his system of patterns to include other contributors he has to be a “system of patterns engineer”. Formalism engineer is a specialization of a patterns engineer.

When creating a pattern for a system of patterns, the engineer plays the role of contributor of this pattern.

Figure 5.5 shows a summary of every type of users we’ve introduced until now.

![Figure 5.5: users in AGAP](image)

**Implementation**

The user component existed already in the previous version, so we enriched it with the new types of users. We also added a DAO for the input and output of this component (chapter 4.2). The user component is extensively used by the administrators which must add/delete/modify users in the system. Here follows some short descriptions of the modifications we’ve made in this component.

**GUI** Figure 5.6 is a screenshot of the interface for creating a user. At the right of the input boxes you can read the validation errors on submissions. You see in this example that there are several mandatory fields to fill in. This verification is performed at the server side by the struts validator (see section 5.6).
Figure 5.6: Creation of a user by users with administrator-rights

Database  The user table in the database had to be adapted to the new types of users. In the previous user table the functions of a user were represented by a string containing the following pattern: XXX where X equals 1 (true) or 0 (false). The first digit represented the function administrator, the second application engineer and the third pattern engineer. This representation was inherited from the first database where Boolean were not implemented. In the new table we are going to replace them by Boolean. A user has 4 types of functions:

- Administrator
- Pattern engineer
- System of patterns engineer
- Formalism engineer

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Not Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>usercode</td>
<td>integer</td>
<td>NOT NULL</td>
<td>PK</td>
</tr>
<tr>
<td>login</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>name</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>surname</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>password</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>society</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>phone</td>
<td>character varying(50)</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>admin</td>
<td>Boolean</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>ingpat</td>
<td>Boolean</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>ingform</td>
<td>Boolean</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>ingsp</td>
<td>Boolean</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
</tbody>
</table>

Table 5.1: The new users table
5.3.4 Group component

Details on publish

Figure 5.7 is a detailed view of the publish mechanism. Only a system of patterns engineer may publish his pattern systems. If the formalism that’s been used in a pattern system hasn’t been exported yet, then the supervisor must be also a formalism engineer to be able to export his pattern system. The formalism must be exported in order to publish the pattern system.

![Figure 5.7: Publishing a pattern system](image)

Now we’ve created access rights for the creation of objects, we can wonder what happens after publication. Because AGAP will be used by several different companies, it’s better to add here also access rights. It is possible that a company doesn’t want that other members that are not part of the company can see the systems they’ve created. Or maybe a company wants to share a system of patterns with an other company in order to cooperate on the same project. Because the systems of patterns are created by a team (the supervisor and the contributors) they have to be able to visualize there creation even after they have been published.

To solve this problem we need the notion of group.

Group component in AGAP

Inside this new component we have 3 types of groups:

**Real group** A Real Group represents a real social group like a group of co-workers, an enterprise, ... This type of group contains other real groups and individuals. The groups are created by the administrator of the AGAP application.

**Group of contributors of a system of patterns** Every system of patterns has a group of contributors. They contain Real Groups or individuals. It’s the responsibility of the supervisor to control this group.
**Group of visitors of a system of patterns** When we publish a system of patterns we create automatically a new group: Group of visitors. This group contains Real Groups and individuals who have the permission to visualise this system. It’s the task of the supervisor to fill this group up at the moment of publication. Note that the group of contributors of this system of patterns is automatically included in the group of visitors. The persons who helped create this system have of course the right to visualise the system after publication.

**Symphony diagram** Figure 5.8 shows a full symphony representation of the new group component. We have an abstract class Group and 3 extensions, resp. the 3 different types of groups (real, contribute and visitors).

In the Symphony representation a component is composed out of 3 parts:

- contract with the outside *(what I can do)*
- structure *(who I am)*
- collaboration *(what I am using)*

**Interface class** This class corresponds to the entrance gate of the component, it describes the operations realised by the component *(what I can do)*.

**Main class** This is the principal class who realises the operations *(what I am)*.

**Subclass** This is a class complementary to the main class *(what I am)*.

**Role class** This class represents a supplier of services next to the client component *(what I am using)*.

---

1 [Ous05]
The group component is new in the AGAP system.

**Action** For the creation of a group we want to use the same page to submit information at the different stages of creation of a group. First stage is giving the NAME and DESCRIPTION. The second stage is filling up with USERS and other GROUPS. To perform this we need to use a lookupdispatcher for the actions. On the createGroup.jsp we have several different submit buttons but only one htmlform. Each of these buttons has a different name to be able to identify the button. We add a parameter to the actionconfig in strutsconfig file. Now when the form has been submitted, Struts looks up which button was clicked and then looks in a map to find the correct action to call.

**Database** For this component we’ve created 3 new tables. The first contains the information of the realgroup concerning the name and description. Each group has also a unique ID. The 2 other tables contain resp. the users and groups.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Not Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>NOT NULL</td>
<td>PK</td>
</tr>
<tr>
<td>name</td>
<td>character</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
<tr>
<td>description</td>
<td>character</td>
<td>NOT NULL</td>
<td>NOT</td>
</tr>
</tbody>
</table>

*Table 5.2: GroupReal table*
**Table 5.3:** Group-group table

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Not Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-groupr</td>
<td>integer</td>
<td>NOT NULL</td>
<td>PK + FK(GroupReal(id))</td>
</tr>
<tr>
<td>id-group</td>
<td>integer</td>
<td>NOT NULL</td>
<td>PK + FK(GroupReal(id))</td>
</tr>
</tbody>
</table>

**Table 5.4:** Group-user table

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Not Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-groupr</td>
<td>integer</td>
<td>NOT NULL</td>
<td>PK+FK(GroupReal(id))</td>
</tr>
<tr>
<td>id-user</td>
<td>character</td>
<td>NOT NULL</td>
<td>PK+FK(Users(Usercode))</td>
</tr>
</tbody>
</table>

**GUI** Figure 5.9 is a screenshot of the GUI for the creation of a Real group. It’s composed out of 2 steps. The first step you fill in the name and description of the group. In the second step (shown in the figure) you can select the users and groups you want to add to this group.

![Screenshot of interface for creating a group](image)

**Figure 5.9:** Screenshot of interface for creating a group

### 5.4 Retracting to sandbox

A system of patterns is composed out of patterns and is expressed in a formalism. The question is now to retract a pattern out of a system of patterns to our sandbox. A pattern can’t exist alone because of the composition. If we stock it only in the sandbox we lose information, in particular the formalism in which the pattern is expressed. To overcome this problem, we proposed the solution to stock in every sandbox a “system of patterns” without a formalism. If we retract a pattern we stock it in that particular system. To prevent the loss of information concerning the formalism, we save a link from the pattern to its formalism.

Thanks to this solution we can now also create patterns without the need of a system. We can save it and it can be used later in a system.
5.5 Symphony diagram

At the end of this chapter we give you the Symphony scheme (fig. 5.13) of the new version of AGAP. In this scheme we have the four components (namely: User, Group, Formalism and System of patterns).

5.6 Implementation

Now that we have analysed the needs of the new system we can start the conception and the development. For this development we’ve followed a certain pattern. We began first with the user component, then we continued with the conception of the new component group and finished with the adaption of the formalism component and the system of patterns component.

5.6.1 Struts

The interaction with the user and the system happens with forms. When the user submits the form, Struts fills up a javabeans and calls the appropriate action. In the application we have a configuration-file named struts-config.xml which contains important information for struts. It contains rules to tell when a form has been submitted which action has to be called. It contains also forward Mappings which tells which JSP has to be loaded. In short, struts works as follows:
1. The user fills in the form and clicks on the submitting button. The HTML formtag contains the name of the action.

2. The action servlet searches for this action. In the action tag there is a description of which action class has to be executed, which actionform has to be used. Struts fills up the actionform and passes it on to the action.

3. The action is executed which can include get and set data in the actionform. At the end the action returns a mapping key to the actionservlet.

4. The actionservlet searches by this key the following JSP to show at the client and runs the JSP.

5.6.2 Validator

We use a validator to prevent the user form filling in the wrong information or from forgetting to fill in some fields. There are 3 ways of using a validator. The first way is to validate, check for errors, when the action is executed. The second way is to let Struts do the validation at the server-side. You create rules for the validation of the form in the validator.xml config file and in the configuration of the action you add validate = true. Now when the form is being submitted, Struts checks it and if there are errors, returns to the form page and outputs the errors. For visualizing the errors you have to add <HTML:errors/> tag in the JSP. The last way to validate a form is to use struts but now on the client-side. To activate it, you configure struts the same as on server side but add a tag to the JSP to load a javascript. Struts will create this javascript.
according to the validator configuration and add it to the JSP. Now you can on submission call the validation function in the script. The errors will be shown in an alert box.

The validator.xml file has the following syntax:

```xml
<validators>
  <field name="someDate">
    <field-validator type="fieldexpression">
      <param name="expression">
        <![CDATA[!someDate.isAfter(@com.domainlanguage.timeutil.Clock@today())]]>
      </param>
      <message>Please enter a date on or before today</message>
    </field-validator>
  </field>
</validators>
```

More information about the configuration and syntax see: [Strb] and a copy of the validator.xml file were using in AGAP can be found in appendix [C]

5.6.3 Error handling

In the current version of AGAP there isn’t any error handling. When there’s a SQLException the user receives a JSP-errorpage that is difficult to understand. Now we’ve created a errorhandling-page. When there is an SQLException, the user will be redirected to the errorpage witch contains the error that has occurred. It offers 2 buttons to the user: “Informing the administrator” of “returning to homepage”. When he clicks on the “inform the administrator”, an email is sent to the administrator to inform him about the error by showing the tracestack. This email function is executed by the mailtaglib ([Mai]) of the struts framework.

![Figure 5.12: Screenshot of the errorhandling in AGAP.](image)
5.6.4 Installation script

Because we have made changes in the database, we need to create a script that automatizes these modifications on the databases when we deliver the new version. The installation script is a JSP page containing several steps (depending on the the modifications). Each time we submit a step, we refer to the same JSP but with the parameter step one higher. To perform database actions we load the PostgresqlDAO class with administrator privileges. The iBatis xml files are stored inside the application depending on the component we’re modifying.
Figure 5.13: The four components of AGAP expressed in the Symphony formalism
Chapter 6

Conclusions

6.1 Results

The AGAP system has now a policy for visualizing and modifying objects. Now, every user has his objects and he only can modify them. It contains also the notion of group which brings it closer to the reality, where users work in teams to create things.

Regarding to the application itself, you can see there are many changes. AGAP has evolved not only from the outside but also from the inside. It contains now an other framework which is used for the I/O to the datasource. The interface has new buttons etc.

6.2 The balance

Generally spoken, this internship was a big experience. It was the first time that I worked in a real enterprise/laboratory environment. It’s completely different from the school environment. It was also the longest project that I have ever worked on.

Thanks to this internship I’ve learned a lot. Before this, I knew a little bit about patterns but not in detail. I had heard about the MVC pattern, but that was it. By helping on the development of AGAP, I learned about patterns, formalisms and pattern systems. I learned about the importance of patterns in applications. Struts and Java Server Pages were also unknown to me, but with the correct information, you get a long way.

During my internship, I spent more time on the analysis and the design of the system than in the actual coding of it. At the beginning I wondered why it was so important and also why I had to design so many UML diagrams. But I’ve learned that thanks to the diagrams I made, I can easily explain to other persons how it works. It helps me also when I was a bit lost in the different concepts. It has also as consequence that the coding goes a lot faster now. Everything has been taught through in advance. This knowledge will help me with future designs.
I can, at the end of this internship, evaluate it positive. The AGAP project is an interesting project with still a lot of possibilities. Working on a project which combines several concepts, from web technology to database technology makes it every time a challenge to let them work correctly.

6.3 Prospects

Now that we’ve implemented the policy of rights for the pattern engineer, we can create the same for the application engineer. This system can make use of keywords and components that we’ve added in this version.

6.4 Planning

This internship took 5 months and at the beginning of it, we had created a planning. Now at the end of those 5 months, I’ve made up the planning realised during the internship. These 2 plannings are represented by a Gantt diagram (figure: and figure: ). The analysis and design took a bit more time than expected.
Appendix A

SQLMapConfig.xml

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    " http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>

<!-- Settings of DB to access them with user privileges-->
<dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="org.postgresql.Driver"/>
    <property name="JDBC.ConnectionURL" value="jdbc:postgresql://sigmacomp.imag.fr/AGAP_DataBase"/>
    <property name="JDBC.Username" value="agapuser"/>
    <property name="JDBC.Password" value="THBU7854"/>
</dataSource>

<!-- Loading SQLMAP with the help of propertiesObject in JAVA -->
<sqlMap resource="${SQLMAP}"/>

</sqlMapConfig>
Appendix B

SQLMAP.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Domain">
    
    <resultMap id="Domain" class="BD.Domain.Domain">
    <result property="NAME" column="domainname"/>
    <result property="ENUMERATE" column="enumeratedtype"/>
    <result property="VALIDATE" column="validate"/>
    </resultMap>
    
    <select id="RecupDomain" parameterClass="java.lang.String" resultMap="Domain">
        SELECT * FROM domain WHERE domainname=#value#
    </select>
    
    <select id="RecupDomainterm" parameterClass="java.lang.String" resultClass="java.lang.String">
        SELECT term FROM domainterm WHERE domainname=#value#
    </select>
    
    <select id="selectValid" resultMap="Domain">
        SELECT * FROM domain where validate='true' order by DomainName
    </select>
    
    <select id="selectNotValid" resultMap="Domain">
        SELECT * FROM domain where validate='false' order by DomainName
    </select>
    
    <select id="selectAll" resultMap="Domain">
        SELECT * FROM domain
    </select>
    
    <delete id="delete" parameterClass="String">
        DELETE FROM domainTerm WHERE DomainName = #value#;
        DELETE FROM domain where DomainName = #value#
    </delete>

</sqlMap>
```
<update id="validate" parameterClass="String">
UPDATE domain SET VALIDATE='true' WHERE DOMAINNAME=#value#
</update>

<insert id="enumerate" parameterClass="String">
INSERT INTO FIELDTYPE VALUES(#value#,'select','','','',','','','','
','','false','true','true')"
</insert>

<insert id="save" parameterClass="BD.Domain.Domain">
INSERT INTO domain VALUES(#NAME#, #ENUMERATE#, #VALIDATE#)
</insert>

<insert id="saveTerm" parameterClass="java.util.HashMap">
INSERT INTO domainTerm VALUES(#domain#, #term#)
</insert>

<update id="update" parameterClass="BD.Domain.Domain">
UPDATE domain(enumeratedtype, validate) VALUES(#ENUMERATE#, #VALIDATE#)
WHERE domainname = #NAME#
</update>

</sqlMap>
Appendix C

Validator.xml

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
"-//Apache Software Foundation//DTD Commons Validator Rules Configuration 1.1.3//EN"
"http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
<formset>
<form name="formalismecreatevalue">
  <field property="NAME" depends="required">
    <arg0 key="app.formalism_nom" />
  </field>
  <field property="AUTHOR" depends="required">
    <arg0 key="app.formalism_author" resource="true" />
  </field>
</form>
<form name="formalismeitemcreatevalue">
  <field property="ITEM_NAME" depends="required">
    <arg0 key="app.item_name" />
  </field>
  <field property="categorie1" depends="required">
    <arg0 key="app.item_cat1" />
  </field>
  <field property="ITEM_DESCRIPTION" depends="required">
    <arg0 key="app.description" />
  </field>
  <field property="ITEM_ORDER" depends="integer,required">
    <arg0 key="app.order" />
  </field>
</form>
<form name="formalismefieldcreatevalue">
  <field property="formalismefield" />
</form>
</formset>
</form-validation>
<field property="FIELD_NAME" depends="required" >
  <arg0 key="app.item_name" />
</field>

<field property="FIELD_TYPE" depends="required">
  <arg0 key="app.fieldtype_name" />
</field>
</form>

<!-- ############# -->
<!-- ADMINISTRATOR -->
<!-- ############# -->
<!-- USERVALUE -->
<form name="uservalue">
  <field property="NAME" depends="required">
    <arg0 key="app.user_name" />
  </field>
  <field property="SURNAME" depends="required">
    <arg0 key="app.user_surname" />
  </field>
  <field property="SOCIETY" depends="required">
    <arg0 key="app.user_society" />
  </field>
  <field property="LOGIN" depends="required">
    <arg0 key="app.user_login" />
  </field>
  <field property="PASSWORD" depends="required">
    <arg0 key="app.user_password" />
  </field>
</form>

<!-- ##### -->
<!-- LOGIN -->
<!-- ##### -->
<form name="loginvalue">
  <field property="LOGIN" depends="required">
    <arg0 key="login.nomvide" />
  </field>
  <field property="PASSWORD" depends="required">
    <arg0 key="password.nomvide" />
  </field>
  <field property="FUNCTION" depends="required">
    <arg0 key="function.nomvide" />
  </field>
</form>
</formset>
</form-validation>
Appendix D

Initial Planning
Appendix E

Actual Planning
Bibliography


