

Database encryption as an Aspect

Gustav Boström
KTH
Electrum 213
164 40 Kista
Sweden
+46-8-7521726
gusbo@kth.se

Abstract

Encryption is an important method for implementing confidentiality in information systems. Unfortunately applying encryption effectively can be quite complicated. Encryption, as well as other security concerns, is also often spread out in an application making implementation difficult. This crosscutting nature of encryption makes it a potentially ideal candidate for implementation using AOP. In this article we provide an example of how database encryption was applied using AOP with AspectJ on a real-life healthcare database application. Although the attempt was promising with regards to modularity, amount of effort and security engineering, it also revealed problems related to substring queries that need to be solved to make the approach really useful.

Keywords

Aspect Oriented Programming, Encryption, Separation of Concern, Java, Database Management, Confidentiality, Security

1. Introduction

Security is generally given as an example of a crosscutting concern that affects large parts of an application. Consequently AOP should be a good technique for implementing security. This has also been proposed by De Win [6], and Viega [19]. However except for DeWin's case study on access control in an FTP-server [5] few case studies on how to do this in practice exist. Access Control is however not the only part of a security implementation that can benefit from AOP. Assuring confidentiality is also an important part of a security implementation that also exhibits the same crosscutting properties as Authentication and Authorization. Confidentiality is often achieved through the use of Encryption. Sensitive data is encrypted to shield it from outside view. The problem is that sensitive data can be spread out over several modules in an application. Everywhere that sensitive data exist the programmer has to remember to explicitly encrypt and decrypt. This is an error prone procedure since it is difficult to remember all places where this procedure should be applied. Furthermore the addition of encryption code to the application leads to greater complexity and code tangling [16]. Encryption in itself can also be a complicated technology to deal with and it is easy to get it wrong [18].

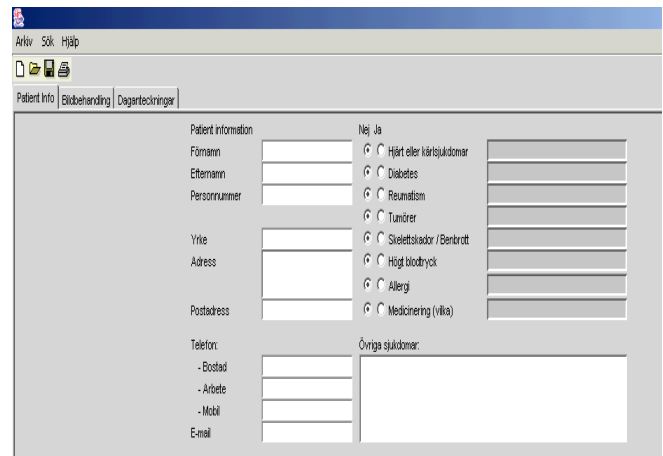
Clearly it would be beneficial if the encryption related code could be factored out of the application and concentrated into one module. If that were the case it would also be easier to give the task of ensuring confidentiality to a dedicated security engineer.

Using AOP it could even be possible to implement encryption without the application programmer even being aware of it.

In this article we demonstrate such a case in which AOP was used to implement encryption in a real-life healthcare application without the initial application programmer having designed the application with this in mind. First we describe the Lumbago Journal application and its confidentiality requirements. Secondly we describe the methodology and tools that were used to construct the solution. Finally we discuss the advantages and problems of an AOP approach compared to traditional implementation methods, the important problem of implementing substring queries is given special attention.

2. The Lumbago application

Lumbago is a simple Java-application used by naprapaths[1] in order to keep a journal of the activities and diagnoses of their patients (Lumbago is the latin name for a common back disorder). It is a small application that consists mainly of two parts: The Patient record and the Day notes. The Patient Record records general data about each patient, such as Social Security Number, diseases, name address and so on.



The screenshot shows a web-based patient record form. At the top, there are navigation tabs for 'Patient info', 'Bleedhandling', and 'Dagarteckningar'. The 'Patient info' tab is active. The form is divided into two main columns. The left column is titled 'Patient information' and contains input fields for 'Förnamn', 'Efternamn', 'Personnummer', 'Yrke', 'Adress', 'Postadress', and 'Telefon' (with sub-fields for '- Bostad', '- Arbeta', '- Mobil', and 'E-mail'). The right column is titled 'Nej Ja' and contains a list of medical conditions with radio buttons for selection: 'Hjärt eller kärlsjukdomar', 'Diabetes', 'Reumatism', 'Tumörer', 'Skellellsjador / Benbrott', 'Högt blodtryck', 'Allergi', and 'Medicinering (vilka)'. Below this list is a section for 'Övriga sjukdomar' with a large text area for additional notes.

Figure 1: The patient record

The Day note part contains the naprapaths notes from a patients visit on a specific day (That is the actual Patient Journal).

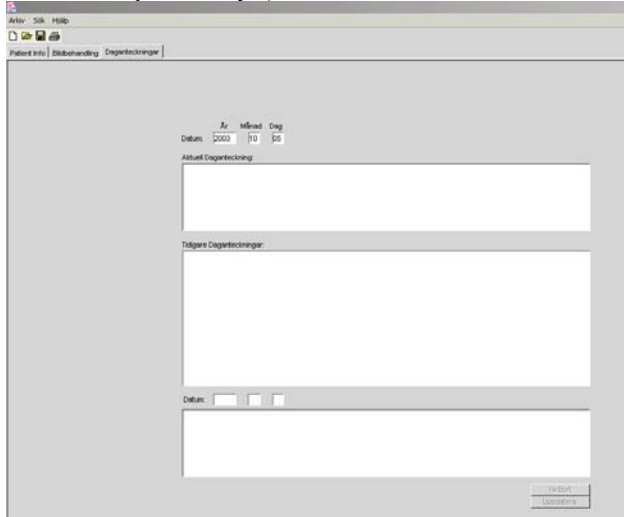


Figure 2: The Journal Form

What makes the Lumbago application unique is that it can also handle store pictures of patients back problems (This however has no impact on the encryption implementation, it is shown here only to give the readers a better insight into the work of naprapaths).

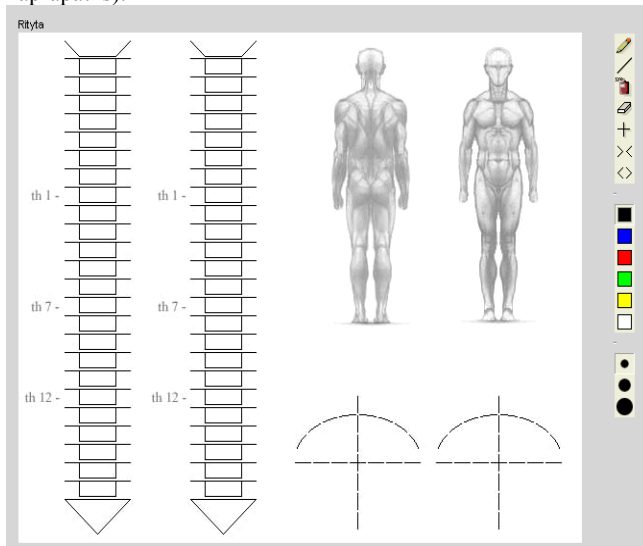


Figure 3: Pain image of the back and vertebrae

2.1 Development organization and development methodology

The application was developed using the eXtreme Programming methodology [3].

The application was developed for a practicing Swedish naprapath, who acted as the XP customer. The developers were undergraduate students working with their own software company on the side. Their intention is to market the application as packaged software.

The encryption aspect was added separately to the application after the initial pre-encryption version had been built. This

development was done by the author, who also acted as the developers' mentor during the project.

Eclipse [7] and the AspectJ-plugin were used as development tools.

2.2 Application architecture

Lumbago was constructed using standard Java. The graphical user interface was built using Swing and the business logic was implemented in plain old Java classes that access a HSQLDB database [12] using direct JDBC [15]. No Object Relational mapping tool was used to map the domain objects to the database since the developers were not familiar with this technology.

2.3 Confidentiality requirements

Healthcare applications have strict confidentiality requirements. Swedish law dictates that any healthcare-application handling sensitive information must deal with privacy issues. The data must be protected so that unauthorized persons should not be able to get at patient journals. The Swedish Social Security Number (The so called "person number".) also has to be protected from outside view. Normally this requires applications to implement access control and also encryption of sensitive data on persistent storage.

Unfortunately this came as a surprise to the unwary Lumbago implementers and the application was not originally built with these requirements in mind.

Consequently login functionality and database encryption had to be added as an afterthought.

3. Method

After the initial version of Lumbago was built (without encryption functionality) the author realized that it could be used for a case study on encryption implementation using AOP. It was especially interesting to use this particular application as a case study since it was not built with encryption in mind. It would put AspectJ[2] and AOP to the test since the original developers were truly oblivious to the encryption aspect (See Filman [8] for a discussion on AOP and obliviousness).

The steps involved for constructing the AOP solution were roughly as follows:

1. Construct a Cryptographer Java-class capable of encryption and decryption
2. Identify where in the application code encryption and decryption needed to be applied, that is, the Encryption Aspect Joinpoints
3. Construct the Encryption Aspect pointcuts for the encryption and decryption joinpoints
4. Add Advice calling the Cryptographer-class at the joinpoints
5. Test the application

3.1 Constructing the Cryptographer-class

The construction of the Cryptographer-class was straightforward. The Java class-library version 1.4 contains the Java Cryptography Extension[14], which can easily be used for encrypting data. In this case a Blowfish symmetric cipher was used. A Junit-test was also constructed to provide a unit test for this functionality.

3.2 Identifying the Joinpoints

Identifying where in the application encryption needed to be applied presented more of a challenge. Considering the application was using plain JDBC [15] to store its data the JDBC-procedure calls seemed a logical place around which you could put the encryption. Encryption should happen before any read or write to persistent storage and the JDBC-calls are just that. These calls are also a good target since they are independent from the business logic. This would make the created pointcuts re-usable in other contexts.

Identifying where encryption and decryption should occur should of course be dependent of what data that needs to be protected. In this case it was decided that for simplicity all strings should be encrypted. Numbers and dates were not considered important to protect (NB, the SSN was encoded as a string).

The joinpoints could be organized in two groups: One for writing/encrypting and the other for reading/decrypting. In the first group you find all calls to `PreparedStatement.setString()`. In the second group you find the calls to `ResultSet.getString()`. However, when examining the Lumbago application closer one sees that instead of `PreparedStatement.setString()`, `PreparedStatement setObject()` was sometimes used. Therefore those calls are also needed as joinpoints. In short one can summarize the writing/encryption joinpoints as:

“All calls to `PreparedStatement.setString()` and `ResultSet.setObject()`”

and the reading/decryption joinpoints as:

“All calls to `ResultSet.getObject()` and `ResultSet.getString()`”.

string, e.g. using the `Statement.execute(String,*)` methods. Luckily these methods were not used in the Lumbago application. It should be possible to handle also these cases by adding also these joinpoints and re-writing the query in the advice. This is a bit more complicated though. JDBC 2.0 also introduces other joinpoints through the use of updateable `ResultSets` that require even more pointcuts to be constructed. In this application these calls were not used. A policy was nevertheless enforced through use of AspectJ *declare*-constructs that issued compile-time warnings when `Statement.execute(String,*)` methods were found in the code. In this way a security engineer can find these places in the code and determine if they are dangerous or not manually (Figure 6).

3.3 Constructing the Pointcuts

This step was merely a translation of the above prose descriptions of the joinpoints into AspectJ syntax. For a novice AspectJ programmer however this can take significant effort. Tool support is essential in order to make this step efficient. Without the support of the `ajbrowser` [2] or the Eclipse AspectJ-plugin this can be a very tedious and error prone step. These tools make it easy to verify that all joinpoints have been included correctly since you can see which methods are affected by advice.

Figure 4 shows an example of a Pointcut for the encryption of strings. When constructing the pointcuts one need not only to indicate the calls that should intercepted, but also you need to bind the data that is needed for the advice around the pointcuts. The data needed in this case is what data that should be encrypted/decrypted. For future use the column name and the `PreparedStatement` is also bound (This should enable encryption

```
pointcut encryptionPointsSetString(String columnName,String value,PreparedStatement stmt):
    ( call(public void PreparedStatement.setString(String,String))
      && args(columnName,value);
```

Figure 4: One of the encryption pointcut definitions

```
Object around (ResultSet result,String columnName):
    decryptionPointsGetObject(result,columnName)
    {
        //Get result and check if it is a string, if so decrypt before returning
        Object object=proceed(result,columnName);
        if(object instanceof String){
            return decrypt((String)object);
        }//Else return unencrypted if not of type String
        ... (Abbreviated)
    }
```

Figure 5: The advice around `ResultSet.getObject()`

As the attentive reader might have guessed these joinpoints are however not enough to ensure that all written data is encrypted. With JDBC it also possible to write to the database by executing SQL-statements in which data has been concatenated in the

to be dependent on which column is written or read).

3.4 Adding advice

Once the pointcuts had been defined we needed to define *what* should happen around these pointcuts. In this case the *around*-advice is well suited, since essentially what needs to be done is to encrypt/decrypt the values before or after the original JDBC-call and replace the argument or result (See Figure 5 previous page). Since Java Strings are immutable it is not possible to do this with *before* or *after* advice. When the `ResultSet.setObject()` are called, the advice also need to find out whether the data is a string or not (in which case encryption should not be performed).

3.5 Testing the application

In order to test the application both unit tests and integration tests were used. For integration testing the original pre-encryption Lumbago test suite was used. This was possible since the Encryption Aspect also affected the unit tests and added encryption to their JDBC-calls. An additional unit test was added to test the Cryptographer class. This enables the encryption functionality to be tested separately.

It is clear that having a test suite is a great advantage. Without a complete test suite you cannot be as confident that adding aspects will not break the existing business logic. The presence of these

would also produce a lot of code tangling since encryption logic will be mixed with database and business logic.

4.1.2 A Design Pattern approach

A better approach than brute force would be to use refactoring and design patterns to improve the separation of concerns[17]. A Proxy[9] could be used around all JDBC read/write calls in order to intercept the arguments and do the encryption/decryption. This is in fact the solution chosen by a French security software vendor company[20]. This would improve modularity a lot. However it would still require the developers to be informed of the reason and the procedure of how to call the proxy instead of the normal JDBC-classes.

Proxies would need to be created for several JDBC-classes, Connection, Statement, PreparedStatement and ResultSet. Considering the amount of methods on these classes this a considerable, but still feasible task. The fact that a lot more code needs to be written for the Design Pattern based approach makes it a lot more risky. All this code needs to be tested as well as written. Hannemann and Kiczales also argue that an AOP implementation of the Proxy problem is both simpler and more modular [9].

```
pointcut manualSQLUpdate():
(
    (call (public * Statement.executeQuery(..))
    || (call (public * Statement.executeUpdate(..))
    || (call (public * Statement.execute(..))
    || (call (public * Statement.addBatch(..))
    );
declare warning: manualSQLUpdate(): "Tried to update database without prepared statement";
```

Figure 6: Pointcut and declare-statements for warning of potentially bypassing statements

tests quickly led to the discovery of the problems with substring LIKE-queries that were introduced by adding encryption.

4. Discussion

Initially the constructed solution looked very promising. Little effort was needed and encryption/decryption was efficiently added to all the applications database write/reads. Unfortunately the problems related to LIKE-queries turned out to be difficult to solve in an easy manner.

In this part we discuss the advantages and problems that were encountered when constructing the solution using AOP.

Several alternative solutions to the encryption problem exist. These are also discussed and compared.

4.1 Alternative solutions

4.1.1 Brute force

The conceptually simplest alternative solution to AOP is probably the “brute force”-method. In this case you just manually add calls to the encryption class in all places that you read/write to the database. However although this solution is conceptually simple this has a very negative effect on modularity since calls to the encryption class would be spread across the application. It

4.1.3 Database encryption

Perhaps the easiest solution would be to let the database handle the encryption aspect. Oracle [13] and probably also other database vendors provide this functionality. It is however not standard functionality and it would therefore tie the application to a particular database vendor. It can also be a considerably more economically expensive option.

A big advantage of the database solution is that for example Oracle can handle searching on encrypted fields. These queries are however orders of magnitude slower than searching on non-encrypted fields. If you have a big dataset this might mean that you will want to restrict querying on these fields anyway [13].

4.2 Effects on modularity

Using the Aspect Visualizer tool in Eclipse gives a picture of where the joinpoints of the encryption points are situated (Figure 7). This also gives a notion of how the encryption aspect would have been spread out over the application had it not been implemented as an aspect. The aspect affected 6 classes out of 31 (Most of the classes are only related to the GUI.). It should be noted that all business logic classes (The domain objects) and many of the unit tests are affected by the encryption aspect (They are affected because they contain JDBC calls for testing the

functionality of the domain objects, so in order for them to work correctly they also need to contain encryption logic when encryption is used).

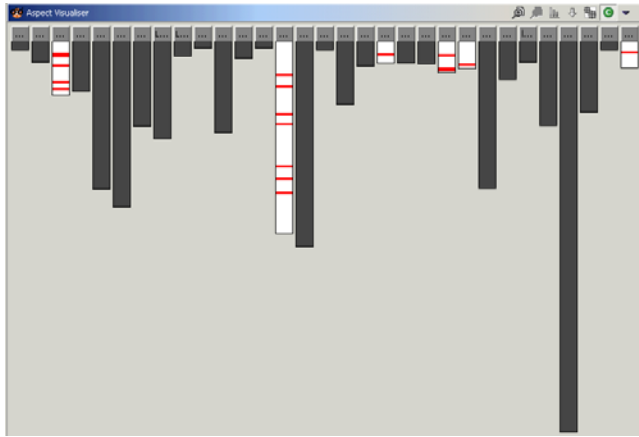


Figure 7: Classes affected by the encryption aspect

Since so many classes would normally have to be re-written because of encryption, it is clear that using AOP improves the modularity. It is important to note that putting the encryption code in a separate class also is a necessary step to improve modularity. Classes, procedures and packages should not be forgotten. AOP is however a particularly efficient way of modularising also the insertion of an aspect, not only it's called logic, that is the server side of the concern.[16]. This is a lot more difficult using only OOP.

4.3 Effects on security engineering

The fact that encryption could be added to the application without the application programmers having prepared for it is a clear advantage from a security engineering point of view. It allows for better separation of concern and therefore a better division of labour between application developers and a dedicated security engineer [4]. It also allows security to be added in a more agile manner since it wasn't necessary to build it in from the start. The fact that AOP can be used to enforce a security policy at compile-time is also an important advantage. Otherwise it would have been more difficult to ensure that unsupported JDBC-methods were not used. Using the compiler for this enforcement is a lot more efficient than code reviews. Use of an encryption aspect could also for example easily enforce a policy mandating that all data passed through JDBC-calls should be encrypted. (For a deeper discussion on regulating architectural decisions using AOP see Shomrat et al [21]).

4.4 Problems with substring queries – a more granular approach needed

A serious problem that was discovered during the testing of the application is that the use of encryption influences the result of SQL-queries in the application. A database field that has been encrypted on the application level cannot be used in a SQL substring query since the encrypted substring will not match (Modern encryption algorithms swap the positions of encoded bytes within the cipher text). The Lumbago application contained functions that allowed the user to do substring LIKE queries, so when the encryption aspect was introduced it broke the search

functionality. This happens since if the user searches for all persons whose first name starts with "Mi", the substring "Mi" would be encrypted to say: "Xz", but in the database the first letters of the encrypted *complete* first name would not start with "Xy" because of the properties of the cipher (It would work with a simple cipher, such as a Caesar cipher, which just substitutes letters, but this hardly provides the accepted confidentiality level).

Either the application would have to be redesigned so that this functionality would be taken away, or encryption would need to be applied on a more granular level. The latter alternative is of course the best one from a usability standpoint. This is however non-trivial. It turned out that it is not easy to find out programmatically which JDBC-calls should be encrypted since it is not always explicit which columns they affect. JDBC-metadata can help in some cases, but not all. It is possible that a query analyzer could be constructed that would allow the Encryption aspect to find this out, but this increases the complexity of the solution to a great extent. This is a serious problem because it invalidates the notion that a developer can remain oblivious of the Encryption aspect. Since encryption affects what business logic that can be performed, business logic and encryption are not independent of each other. A simple, but not so elegant, solution to this problem is to add extra pointcut definitions that point out the joinpoints where encryption should not be performed. The drawback with this solution is that it couples the Encryption aspect very tightly to the application. It is also possible that if these pointcuts are not precise enough they will affect the wrong places when the application evolves. In this application this was not even possible since the specific JDBC-calls that needed to be excluded from encryption advice could not be easily identified by pointcuts. The only way was to refactor the code and put these calls in separate methods so that they could easily be defined in a pointcut.

4.5 Would Object-relational mapping be easier?

The problem of achieving granularity of the encryption using a JDBC-aspect leads one to think that it would have been easier to implement the encryption if the application would have been constructed using some OO-mapping software, such as Hibernate[11], instead. This would be easier since the getter- and setter-methods of the classes will provide explicit granular joinpoints. An OO-mapping framework would also limit the amount of possible places where encryption needs to be added, since it wraps the database in a simpler interface. However it could also prove more difficult, since it is then necessary to understand and modify the inner workings of the persistence framework.

4.6 Familiarity with AOP

Another important problem of a different sort was that the AOP approach was not used in production due to the fact that the original application developers did not have time to learn AOP and therefore did not feel comfortable with its use. This is probably a problem with most new tools though. Time has to be invested in order to make developers familiar with the environment.

5. Conclusion

Implementing application-level encryption using AOP is promising since it results in better modularity, database independence and less code. There are however still some important problems that need to be solved in order for the approach to be really useful. If there is a requirement on the application to perform substring queries special handling needs to be done and the business logic developers can no longer be oblivious of encryption. This invalidates an important advantage of an AOP approach. A more fine granular approach where encryption can be applied selectively at column level is needed. In other situations where all data can be indiscriminately encrypted the solution should work better. This situation is however probably rare in larger projects.

6. Suggestions for further work

Although the results in this case study are promising the substring query problem needs to be solved before an encryption aspect can be really useful. A query analyzer that will enable a more fine-grained application of encryption could be a solution.

An easier approach would probably be to redesign the application to use an Object-relational mapping framework. That possibility needs to be more thoroughly investigated however, since it might reveal other problems.

The encryption aspect in this application became tightly coupled to the business logic. It would be useful to abstract out the general parts to an aspect framework that could be extended and used also in other JDBC applications.

7. Acknowledgements

The author would like to thank the Lumbago project team for letting us use the Lumbago application for research purposes. Thanks are also due to the AOSD 2004 Practitioner Report Committee for providing useful comments on how to improve the article.

8. REFERENCES

- [1] An explanation about naprapathy, <http://www.rsdcrps.com/naprapathy.html>
- [2] AspectJ, <http://eclipse.org/aspectj/>
- [3] Beck K., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000
- [4] B. De Win, F. Piessens, W. Joosen and T. Verhanneman, On the importance of the separation-of-concerns principle in secure software engineering, Workshop on the Application of Engineering Principles to System Security Design, 2002, workshop reader will be published
- [5] B. De Win, W. Joosen and F. Piessens, AOSD & Security: a practical assessment, Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03), 2003, pp. 1-6
- [6] B. De Win, B. Vanhaute and B. De Decker, How aspect-oriented programming can help to build secure software, *Informatica* vol.26(2), 2002, pp. 141-149
- [7] Eclipse, www.eclipse.org
- [8] Robert E. Filman and Daniel P. Friedman. "Aspect-Oriented Programming is Quantification and Obliviousness." Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.
- [9] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [10] Jan Hannemann, Gregor Kiczales: Design pattern implementation in Java and aspectJ. OOPSLA 2002: 161-173
- [11] Hibernate, www.hibernate.org
- [12] HSQLDB, www.hsqldb.org
- [13] Implementing Data Encryption, <http://www.interealm.com/technotes/robby/encrypt.html>
- [14] Java Cryptography Extensions (JCE), java.sun.com/products/jce/
- [15] Java Database Connectivity, java.sun.com/products/jdbc/
- [16] Laddad, R, *AspectJ in Action*, Manning Publications, 2003
- [17] Kerievsky, J, *Refactoring to Patterns (Draft)*, <http://www.industriallogic.com/papers/rtp017.pdf>
- [18] John Viega, Gary McGraw. *Building Secure Software*. Addison Wesley. Summer, 2001.
- [19] John Viega, David Evans. Separation of Concerns for Security. In ICSE Workshop on Multidimensional Separation of Concerns in Software Engineering, June 2000.
- [20] SafeJDBC, <http://www.safejdbc.com/index.html> (In French)
- [21] Shomrat, Mati Shomrat, Amiram Yehudai: Obvious or not?: regulating architectural decisions using aspect-oriented programming. AOSD 2002: 3-9