Generation of an Architecture View for Web Applications using a Bayesian Network Classifier

Juan-Carlos Castrejon-Castillo, Rafael Lozano, Genoveva Vargas-Solar

Generation of an Architecture View for Web Applications using a Bayesian Network Classifier

Juan Castrejón, Rafael Lozano
Campus Ciudad de México
Instituto Tecnológico y de Estudios Superiores de Monterrey
{A00970883, ralozano}@itesm.mx

Genoveva Vargas-Solar
LIG-LAFMIA Labs
Centre National de la Recherche Scientifique
Genoveva.Vargas@imag.fr

***Abstract*—A recurring problem in software engineering is the correct definition and enforcement of an architecture that can guide the development and maintenance processes of software systems. This is due in part to a lack of correct definition and maintenance of architectural documentation. In this paper, an approach based on a bayesian network classifier is proposed to aid in the generation of an architecture view for web applications developed according to the Model View Controller (MVC) architectural pattern. This view is comprised of the system components, their inter-project relations and their classification according to the MVC pattern. The generated view can then be used as part of the system documentation to help enforce the original architectural intent when changes are applied to system. Finally, an implementation of this approach is presented for Java based-systems, using training data from popular web development frameworks.**

***Index Terms*—Documentation, Software engineering, Software maintenance, Software verification and validation**

I. Introduction

The use of a software architecture to guide the development and maintenance processes of a system has been widely studied in the software engineering area [1]. Even though there is not a unique definition of software architecture, the general idea is that it should describe the system components along with their relations, recognizing that there is not a unique representation that can comprise all these relations and components [2]. It is for this reason that architecture views are created to support the different representations of the system components, their relationships and requirements. Each of these views can then be used to communicate requirements and design constraints to the system stakeholders [1].

One or more architecture patterns are usually taken as reference models for the definition of the components types that constitute a software solution [1]. These patterns convey common structures and interactions that are proved to solve particular requirements. It is important to note that, in order to be useful, the patterns intents should be documented in one or more of the system architecture views.

Concerning web application development, the *Model View Controller* (MVC) pattern [3] is one of the most influential architecture patterns that have guided design and implementation of web systems for the last years [4], due to its natural separatation of business, presentation, and control logic. This separation greatly reduces the complexity needed to apply changes to individual components, by maintaining clear responsibilities and dependencies by means of three logical grouping layers, that is, *Model*, *View* and *Controller* layer.

The *Model* layer contains the business information with which the system operates. The *View* layer is responsible for the presentation of the information in a way suitable to the system users. The *Controller* layer is in charge of responding to events, probably invoquing changes both in the *View* and *Model* layer. Web applications usually implement a variation of the MVC pattern, named *Model 2* [5]. This variation focuses on efficiently handling and dispatching full page form posts and reconstructing the full page content via front controllers.

In regard to the classification of system components, we should consider that a classification process can be defined as a formal method used to establish a function whereby we can associate a new observation into one of existing classification groups [6]. This process is also known as *Supervised Learning*, and allows us to deduce the classification function from a set of training data, consisting of input objects and desired outputs.

Numerous approaches to the problem of generating a function classifier from a set of training data are based on the Bayes' theorem [6]. This theorem expresses posterior probability of a hypothesis in terms of previous probability of the hypothesis given evidence, and the probability of the evidence given the hypothesis. Bayes' theorem is shown in the following equation, considering the variable A as the hypothesis and variable B as the evidence.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

A Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional independences using a directed acyclic graph [6]. The use of this probabilistic model as classifier allows us to benefit from the results of a training process in which the classification groups are identified from the analysis of key random variables.

In this case, it is important to notice that given a sufficiently large dataset of training instances, the resulting probabilistic model will represent a close approximation to the probability distribution governing the random variables domains [7].

II. Related Work

The use of Bayesian approaches for modeling the uncertainty in software systems has been studied previously along with the so-called *Uncertainty Principle in Software Engineering* [8]. This principle states that uncertainty is inherent and inevitable in software development processes and products. It is suggested that uncertainties associated with one or more properties of software artifacts should be modeled and maintained explicitly. To that purpose, the use of networks of software artifacts, annotated with uncertainty values, can be used to effectively analyze the uncertainty of the system as a whole, supporting prediction and guidance of future development activities [8].

The use of probabilistic models that analyze the source code of software systems in order to identify clusters, is a wide spread technique among reverse engineering tools [9], [10]. In this regard, a probabilistic approach is described in [11] for the partition of software systems into meaningful sub-systems, using the information of *zones* inside each compilation unit. This approach includes the analysis of method and class signatures, along with variable identifiers. These classification variables are similar in nature to the ones used in this study, however the classification mechanisms differ.

In [12] an approach based on Bayesian learning is proposed to automatically recover a software system's architecture, given incomplete or out-of-date documentation. The proposed methodology includes the training of a Naïve Bayes classifier that is later used to predict an appropiate sub-system for the project software modules. The global variables accesed by these software modules are then used as attributes for the bayesian model. This represents a key difference with the approach presented in this paper, because we take into consideration dependencies not only through global variables, but also through external APIs and inter-project associations.

III. Problem

According to the *Uncertainty Principle in Software Engineering*, uncertainty permeates software development in all of its phases [8]. Coupled with inherent complexity in software systems, software quality and developer productivity are topics that require monitoring to avoid degrading over time.

Monitoring of a correct implementation of architecture patterns during the development and maintenance of a software system is not always enforced by the project members [13]. This is one of the reasons why a system may degenerate from the initial architecture intent, causing severe problems when changes have to be applied to the system.

We also have to consider that for the implementation of architecture patterns, uncertainty arises because we do not know for sure to which code elements the patterns components are mapped. Ideally, this information would be kept in the project's documentation, but in a great number of software systems, it is unavailable, incomplete or out-of-date [13]. A reason for this situation is the fact that software systems are expected to undergo a number of changes during their lifetime, and even if architectural documentation was developed for the original system, there's no guarantee that this documentation was effectively updated to reflect subsequent changes [13].

In this regard, a key feature in the domain of Web applications is the separation of responsibilities for maintaining each of the project components [3]. The MVC pattern promotes this separation by assigning project components to one of three possible grouping layers, and defining clear interactions among them. However, a problem arises when an implementation does not quite follow the architectural intent of the patterns being used [13]. For instance, when a component of a given layer is given more responsibilities than it should, or when invalid dependencies are added between components. As the system undergoes new changes, these incorrect implementations add complexity to the maintenance of the project, probably causing a detriment to the system quality.

The process of updating software system documentation can be misleading without the appropiate mechanisms to validate that implementation changes comply with the architectural constraints imposed over the project. Ultimately, this mechanisms should enforce that the original architectural intent is maintained over the project's life cycle.

IV. Objectives

An approach with the following objectives is proposed in order to help avoid the problems stated in the previous section:

- Classify each of a web project components into one of the layers defined by the MVC pattern.
- Analyze the relationships among components to help identify invalid dependencies.
- Generate an architecture view comprised of the project components and their inter-project dependencies.
- Provide an implementation of this approach that can be used in Java web projects.

Considering that web application development covers a wide range of technologies [5], it would be very ambitious to develop an implementation to support all of them. Nonetheless, in order to provide support for best practices and the general approach followed in Java web application development, the implementation described in this paper considers four of the most popular frameworks in the Java environment. These are:

- Grails [14].
- Spring Roo [15].
- Play framework [16].
- Apache Struts 2 [17].

V. Proposed solution

A. Uncertainty Model

To handle the uncertainty in the implementation of the MVC architecture pattern, the use of a Bayesian network classifier is proposed. This choice has been made because this probabilistic model allows us to represent in an effective manner the conditional dependencies between random variables. In this case, the variables are based on the system implementation artifacts and its relationships with components defined outside the project, that is, external libraries. Using this classifier, the project components are grouped according to the MVC layers.

The following random variables, and their domains, are proposed in order to generate an effective probabilistic model:

- *Type*. Identfies a component's file type (domain: *java*, *jsp*, *xml* and *html*).
- *Suffix*. Identifies the suffix of a component's file name (domain: *controller*, *service*, *validator*, *context*, *servlet*, *web*, *aspect*, *form*, *dao*, *manager* and *none*).
- *ExternalAPI*. Identifies the external API that a component depends on (domain: *springmvc*, *aspectj*, *hibernate*, *jdbc* and *none*).
- *Layer* . Identifies the grouping layers defined by the MVC pattern (domain: *model*, *view* and *controller*).

The intent of using these random variables is to capture a component's behaviour through the analysis of standard naming conventions and the identification of common external APIs used to carry out the expected responsabilities of components in each of the layers defined by the MVC pattern.

For instance, a component grouped in the *Controller* layer is most likely to depend on the *Spring MVC framework*, and less likely to depend on database access APIs, such as *JPA* or *Hibernate*. We assume a similar logic for the naming conventions. Take for example a controller component, it is more likely to be named **Controller.java*, than **Dao.java*.

To complete the definition of the Bayesian network, we need to identify dependencies among the random variables, to later assign probabilistic values to the possible combinations of these variables. One way of doing this is by using historical data, to obtain an approximation of the probability distribution governing the attributes domains. For this case, we used the data of *17* representative web projects that make use of the Java based frameworks mentioned in the previous section. These sample projects are included in the public distributions of these frameworks, and are listed next:

- Grails: *Recipes* [14].
- Spring Roo: *Pet Clinic, Vote and Wedding* [15].
- Play framework: *Booking, Chat, Forum, Job Board, Twitter, Validation, Yabe and Zen* [16].
- Struts 2: *Blank, Mail reader, Portlet, Showcase and Rest Showcase* [17].

Altogether, these applications were comprised of *619* components that had to be manually analyzed and classified. In order to obtain the dependency relations and probabilistic values for all the possible combinations of the random variables, the Weka software [18] was used, taking as input this manually classified set. The complete process for creating the probabilistic model using Weka is detailed next.

- An *Attribute-Relation File Format* file [18], ARFF, was created to store the manual classification results.
- Each of the applications components was manually analyzed in order to assign values to the *Type*, *Suffix*, *ExternalAPI* and *Layer* random variables.
- The Weka Explorer interface was used to classify the data in the ARFF file according to the *BayesNet* Classifier using the *TAN* search algorithm [18].
- The probabilistic model was saved into a *Model object file* [18]. This model is later used during classification.

After this process, Weka generates a probabilistic model, depicted in Fig. 1, along with the probability distribution tables for all the possible combinations of the random variables. For illustrative purposes, the distributions tables for the *Layer* and *Types* variables are shown in Tables I and II, respectively.

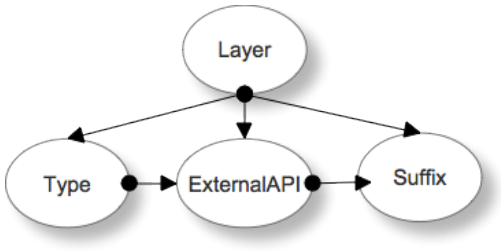

Fig. 1. Weka-generated Bayesian model

TABLE I
PROBABILITY DISTRIBUTION TABLE FOR THE LAYER VARIABLE

Model	View	Controller
0.28	0.257	0.463

TABLE II
PROBABILITY DISTRIBUTION TABLE FOR THE TYPE VARIABLE

	File Type			
Layer	java	jsp	xml	properties
model	0.986	0.003	0.003	0.009
view	0.034	0.562	0.158	0.245
controller	0.642	0.002	0.317	0.04

Using the Bayesian network generated by Weka, *542* instances of the ARFF file are corrected classified. That is, we have around *87%* of effectivity using the training set as a test option. This high effectivity is compelling enough to think that this model provides a relatively good approximation to the probability distribution governing the attributes domains. In Fig. 2 the Weka classification results are shown:

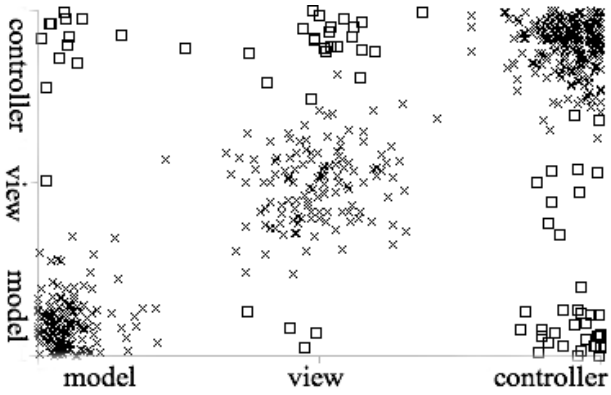

Fig. 2. Weka classification results. Please note that a *square* represents an error, while a *cross* represents a successful classification.

Another advantage of using Weka for the generation of the probabilistic model is that the more data we gather the better our probability distributions will be, without us needing to manually update the values in the probability distribution tables. We would just need to repeat the process explained before, in order to generate an updated probabilistic model.

B. Relationship analysis

Complementing classification results, a further analysis is perfomed to guarantee that invalid dependencies, as stated in the *Model 2* pattern [5], a variation of MVC for web projects, are not found in the implementation components. This analysis also verifies that the grouping of components by packages and directories is consistent with a traditional web project distribution. The *Model 2* pattern [5] is depicted in Fig. 3.

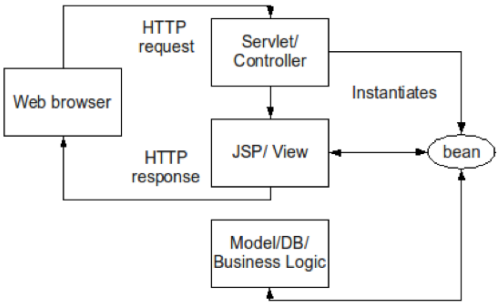

Fig. 3. Model 2 pattern

A component classified as *Model* should only have relations with other *Model* components. A component classified as *View* shouldn't have direct relationships with *Model* components, instead the *Controller* components should provide the required Java Beans that contain a subset of the model for the *View* components to use. A component classified as *Controller* can have direct relations with both *Model* and *View* components. Those components that don't conform to this relationship rules are marked as invalid in the generated architectural view.

Once the components are classified, all the packages that contain code artifacts are analyzed to get the most frequent component type for each of the packages. For example, a package containing a majority of *View* components is classified as a *View* package. Once all the packages are classified, those components having a different component type than that of their package, are marked as invalid. A similar logic applies for directories that does not necessarily contain code artifacts.

C. Architecture view

As final result, an architecture view is generated by grouping the classified components according to their package or directory, showing for each component its MVC classification and by linking the components using their inter-project relationships. The components identified as invalid during the Relationship analysis are highlighted to allow further examination by the system development team.

D. Implementation for Java based projects

An implementation for Web projects, supporting a subset of the technologies used in the Java environment is provided to help test the validity of the proposed approach [19].

The implementation includes a generic module that can be included in any development environment, by means of JAR file import. A second module includes plugins for the Eclipse platform [20], to promote an automated use of our approach.

In order to use the Bayesian network, previously generated with Weka, in this implementation, we need to be able to assign for each of the analyzed components, values for the *Type*, *ExternalAPI* and *Suffix* random variables. In this case, the assignment is done through a static code analysis process.

For code components, the dependencies evaluation part of the static analysis includes the use of the ASM framework [21]. Based on the *Visitor* design pattern [3], a new class was developed to extend from ASM's *EmptyVisitor* class, to allow us to inspect a project's Java classes. This new class, *DependencyVisitor*, is responsible for identifying for each class both its inter-project and external dependencies.

The drawback of this approach is that classes used by an analyzed component that belong to the same package, are not added to the inter-project dependencies, unless they're declared as argument of a method. However, this does not affect the final results because of the Relationship analysis that's perfomed after the components classification.

As a result of this dependencies evaluation, we can give values to the *ExternalAPI* variable, by looking for supported libraries in the components' external dependecies list. For Java classes, the *Type* variable is assigned the value java. The *Suffix* variable can be given values directly from the java classes filenames. For non-code components, only the *Type* and *Suffix* variables are considered before classification. Values are obtained directly from the components file names.

Once the components have been assigned values for the *Type*, *Suffix* and *ExternalAPI* variables, the Bayesian network model is loaded from disk and is then used to classify those components, thus giving values to the *Layer* variable.

Once the classification process is over, the Relationship analysis is performed for all the project components, in order to identify invalid relations among them. It should be noted that the definitions of valid relationships are kept in the definiton of the *Layer* class. Instances of this particular class are then associated to the project components, according to the result of their classification process.

If an invalid relation is identified for a component, it is marked as invalid. Project packages and directories are classified into one of the MVC layers by grouping the components associated to them according to their classification, and then taking the grouping layer of the largest group. Those components having a different classification than that of their package or directory are also marked as invalid. Once all the components are classified and the Relationship analysis is over, a SVG file is created from this data using the Graphviz drawing tools [22]. Layers are identified by a combination of a color and a shape. These combinations are shown in Fig. 4.

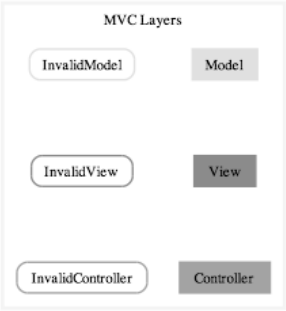

Fig. 4. Graphic representation of the MVC Layers

We can see that each layer has its invalid counterpart, that differs from the valid one in shape and fill style but not in color. These are the styles used in the architecture view.

E. Eclipse plugin

The implementation details explained in previous sections are part of a generic implementation module. Besides this module, a plugin for the Eclipse platform was developed to ease the use of this implementation in Java projects. This plugin [19] allows us to invoke the MVC classifier process from within an Eclipse view, and also see the results within Eclipse. The plugin is designed to work with Project elements and WAR files, by adding a context menu entry when these objects are selected with a *right-click* in an Eclipse view.

The Eclipse menu provided by our plugin is represented by the *Pattern Views –>Generate MVC View for Project-War files* path. An instance of this menu is depicted in Fig. 5.

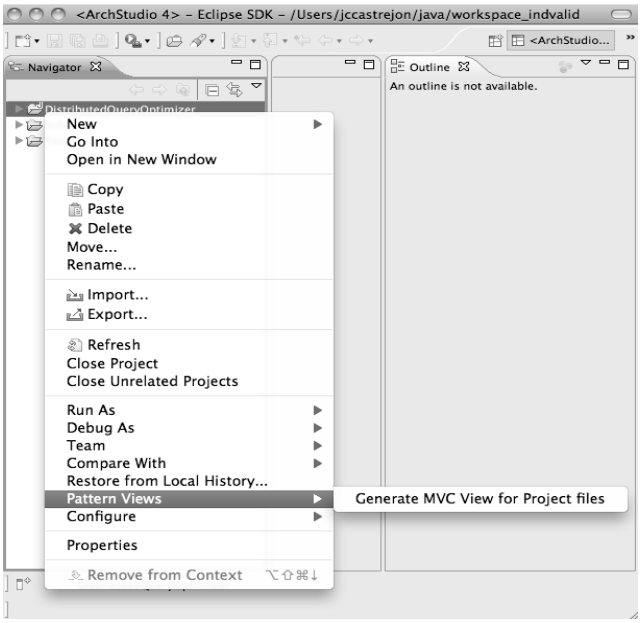

Fig. 5. Context Menu for Project elements

Finally, if any of the context entries is selected, an Eclipse view containing the generated architecture view is shown to the user, using the SVG Salamander library [23] (See Fig. 6).

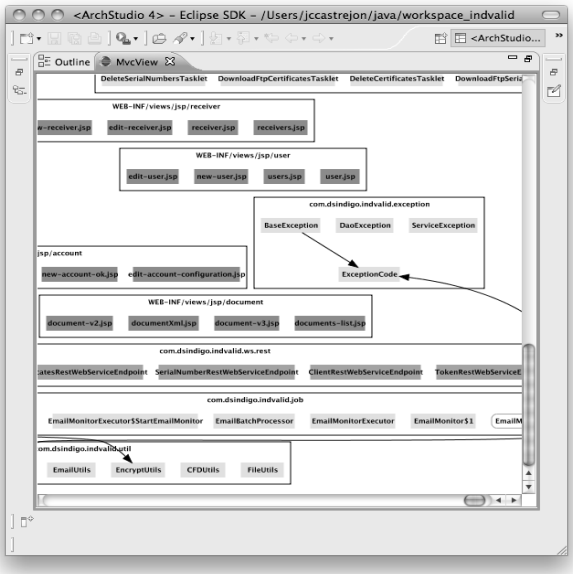

Fig. 6. *MvcView* Eclipse view

F. Sample application

In order to explain the use of the Java implementation described in the previous section, we now present fragments of the architecture view generated for a real-world Java web application. This application is named *Indvalid* [24] and was designed to automatically validate digital tax receipts, in order to help enterprises comply with the mexican regulation in this regard. This application is a Java web system developed using several modules from the Spring Framework [25].

To show how valid components are depicted in the architecture view, we can analyze the following fragment in Fig. 7.

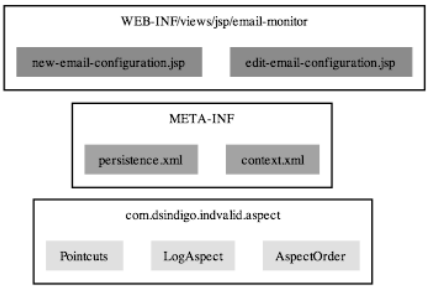

Fig. 7. Fragment of an architecture view showing valid components

By analyzing this fragment of the architecture view, we can see that components are grouped by packages and directories, according to their file type. For each component, its classification results are shown by means of its shape and color. Each layer is represented by at least one component. Note that only valid components are shown in this architecture fragment.

In order to show how invalid components are depicted in the architecture view, a fragment of a package contained in the view generated for the *Indvalid* system is depicted in Fig. 8.

Fig. 8. Fragment of an architecture view showing invalid components

We can appreciate that two components of this package have been marked as invalid. This is because their classification group, *Model*, differs from the package's classification group, *Controller*. This package was classified as such because most of the components it contains were classified as *Controller*.

Finally, to show how a more complete architecture view is presented to the user, a bigger part of the architecture view generated for the *Indvalid* project is depicted in Fig. 9.

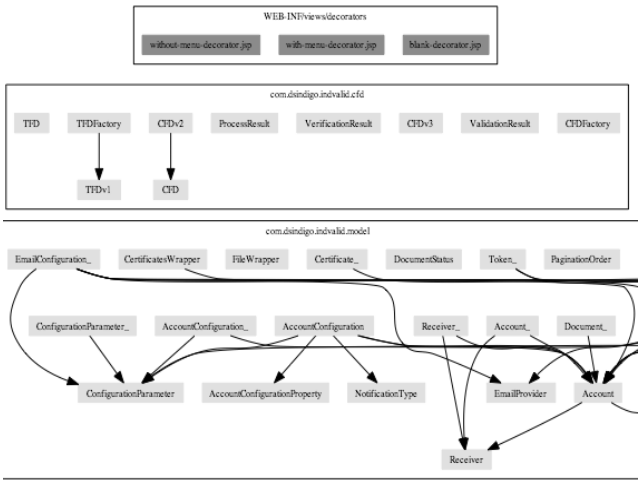

Fig. 9. Fragment of the *Indvalid* architecture view

VI. Conclusions

In this paper, an approach for the generation of an architecture view was proposed for MVC Web applications, by using a bayesian network classifier over the components that comprise a software system, and by identifying invalid relationships among them. The probabilistic model, a bayesian network, is generated using the Weka software, and trained with data of a representative set of sample web applications associated to popular Java based web frameworks. In this first stage, the effectiveness of the generated probabilistic model is promising.

In order to validate the proposed approach, an implementation for Java Web projects was developed. This implementation is divided in two parts, a generic module and an Eclipse plugin. Using this implementation, development teams will be able to use the classifier with any Java web project, generating the architecture view into a SVG file, allowing further analysis.

The generated architecture view can be used as a complement to the existing documentation associated to web applications. If no documentation exists, this view may represent a good starting point for development teams in order to help identify the system components, their responsabilities and interactions between them, all based on the MVC pattern.

VII. Future Work

In order to improve the accuracy of the model, the bayesian network should be trained and tested with data covering a wider range of web projects. To this extent, the implementation should support not only Java but also other web technologies.

An improvement to the proposed approach could be the generation of the architecture view not only as a SVG file, but also using standard architectural notations, such as UML.

The probabilistic model may also be updated with additional random variables in order to perform a more thorough analysis of the components associated to web projects. Finally, the classification of system components with alternative algorithms and data mining tools is also intended as future work.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. New York: Addison-Wesley Professional, 2 ed., 2003.
[2] I. Gorton, *Essential Software Architecture*, ch. Understanding Software Architecture, pp. 1–15. New York: Springer, 1 ed., 2006.
[3] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.
[4] Sun-Developer-Network, "Java blueprints - j2ee patterns." http://java.sun.com/blueprints/patterns/MVC-detailed.htm/, July 2011.
[5] Sun-Developer-Network, "Designing enterprise applications with the j2ee platform." http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html, July 2011.
[6] R. E. Neapolitan, *Learning Bayesian Networks*. Prentice Hall, 2003.
[7] F. Jensen and T. Nielsen, *Bayesian Networks and Decision Graphs (Information Science and Statistics)*. Springer, 2 ed., 2007.
[8] H. Ziv and D. Richardson;, "The uncertainty principle in software engineering," in *ICSE'97, 19th International Conference on Software Engineering*, (Boston, MA), 1996.
[9] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, pp. 759–780, November 2007.
[10] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.
[11] A. Corazza, S. D. Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 88–96, 2010.
[12] O. Maqbool and H. Babri;, "Bayesian learning for software architecture recovery," in *Electrical Engineering, 2007. ICEE '07. International Conference on*, (Lahore, Pakistan), 2007.
[13] J. Chen and S. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software*, vol. 82, no. 6, pp. 981–992, 2009.
[14] SpringSource, "Grails." http://grails.org/, July 2011.
[15] SpringSource, "Spring roo." http://www.springsource.org/roo, July 2011.
[16] Zenexity, "Play framework." http://www.playframework.org/, July 2011.
[17] Apache-Software-Foundation, "Apache struts 2." http://struts.apache.org/2.x/, July 2011.
[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.
[19] J. Castrejón, "Bayesian network mvc analyzer." http://code.google.com/p/mvc-analyzer/, July 2011.
[20] Eclipse-Foundation, "Eclipse-ide." http://www.eclipse.org/, July 2011.
[21] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: A code manipulation tool to implement adaptable systems," in *In Adaptable and extensible component systems*, 2002.
[22] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, "Graphviz open source graph drawing tools," in *Graph Drawing* (P. Mutzel, M. Jnger, and S. Leipert, eds.), vol. 2265 of *Lecture Notes in Computer Science*, pp. 594–597, Springer Berlin / Heidelberg, 2002.
[23] M. McKay, "Svg salamander." http://svgsalamander.java.net/, July 2011.
[24] Indigo-IT, "Indvalid." http://www.indvalid.com/, July 2011.
[25] SpringSource, "Spring framework." http://www.springsource.org/about, July 2011.