

# Generation of an Architecture View for a MVC Web application through the use of a Bayesian Network Classifier

Juan Carlos Castrejón Castillo

**Abstract**—A recurring problem in software engineering is the correct definition and enforcement of an architecture that can be used to guide the development and maintenance processes of a software system. 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 MVC pattern. This view is comprised of the system components, their inter-project relations and their classification according to the MVC architecture 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 Web projects.

**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, and recognize 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 software requirements and design constraints to the system stakeholders [1].

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

Concerning web applications, the MVC architecture pattern has been the one with the greater influence over the way systems have been designed and built for the last years [3]. This pattern helps us to separate business logic from 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, 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 [4]. This variation focuses on efficiently handling and dispatching full page form posts and reconstructing the full page via a front controller.

A classification process can be defined as a formal method used to establish a function whereby we can classify a new observation into one of existing classification groups [5]. This process is also known as Supervised Learning. The use of this process 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 [5]. 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. That is, considering  $A$  as the hypothesis and  $B$  as the evidence:

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

Fig. 1. Bayes' Theorem

A Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional independences using a directed acyclic graph [5]. The use of this probabilistic model as classifier allows us to benefit from the results of a training process in the search of the classification group for a new set of attributes values, such that this group maximizes the posterior probability from the set of classification groups, identified during the training phase. Given a large dataset for training, the result model will be a close approximation for the probability distribution governing the attributes domains [6].

### A. Work related

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* [7]. 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 [7].

In [8] 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 appropriate subsystem for the project software modules. The global variables accessed by these software modules are used as attributes for the bayesian model.

## II. PROBLEM

According to the Uncertainty Principle in Software Engineering, uncertainty permeates software development in all of its phases [7]. Coupled with inherent complexity in software systems, the software quality and developer productivity are usually affected.

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 [9]. 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 [9]. 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 updated to reflect subsequent changes [9].

A key feature in the domain of Web applications is the separation of responsibilities for maintaining each of the project components [10]. The MVC pattern promotes this separation by assigning project components to one of three possible grouping layers, and defining clear interactions among them. A problem arises when an implementation does not quite follow the architectural intent of the patterns being used [9]. 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 appropriate 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.

## III. OBJECTIVES

The main objectives of the proposed approach are:

- 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 dependencies.
- Provide an implementation of this approach to be used in Java web projects.

Considering Java Web development covers a wide range of technologies [4], it would be very ambitious to develop an implementation to support all of them. For this reason, at this first stage, the implementation of the approach for Java Web Projects considers only the following technologies:

- JSP files
- XML files
- HTML files
- Java classes
- Basic Support for the Spring Framework
- Basic Support for the Hibernate Framework

## IV. 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 will be based on the implementation components and its relationships with components defined outside the project, external libraries. Using this classifier, the project components will be grouped according to the MVC layers. The random variables proposed for the probabilistic model are detailed next.

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

The intent of 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 responsibilities of components in each of the layers defined by MVC.

For instance, a component grouped in the Controller layer is most likely to depend on the Spring MVC framework, and less likely to use the JDBC API. 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 domain. For this case, we used the data of representative web projects that make use of the previously stated supported technologies for the Java environment. These projects are:

- The PetClinic application, included in the Spring Framework documentation [10].
- The MVC Step by Step tutorial, also included in the Spring Framework documentation [10].
- The Spring MVC tutorial by [11].
- A Distributed Query Optimizer application by [12].

Altogether, these applications were comprised of 114 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 [13] was used, taking as input this manually classified set. The process of creating the probabilistic model using Weka is detailed next.

- An Attribute-Relation File Format file [13], ARFF, was created to store the manual classification results.
- Each of the applications components was 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 [13].
- The generated probabilistic model was saved into a Model object file.

The probabilistic model generated by Weka, a Bayesian Network, is depicted in the following Figure:

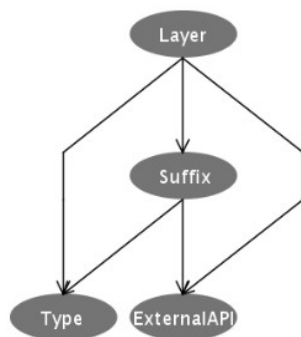


Figure 2. Bayesian Network

During this process, Weka also generates the probability distribution tables for all the possible combinations of the random variables. The values for these tables are shown next:

	model	view	controller
	0.385	0.29	0.325

Figure 3. Probability Distribution Table for the *Layer* variable

Layer	controller	service	validator	context	servlet	web	aspect	form	dao	manager	none
model	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.051	0.01	0.859
view	0.013	0.013	0.013	0.013	0.013	0.013	0.013	0.091	0.013	0.013	0.792
controller	0.129	0.106	0.153	0.059	0.106	0.129	0.082	0.012	0.012	0.059	0.153

Figure 4. Probability Distribution Table for the *Suffix* variable

Layer	Suffix	java	jsp	xml	html
model	controller	0.25	0.25	0.25	0.25
model	service	0.25	0.25	0.25	0.25
model	validator	0.25	0.25	0.25	0.25
model	context	0.25	0.25	0.25	0.25
model	servlet	0.25	0.25	0.25	0.25
model	web	0.25	0.25	0.25	0.25
model	aspect	0.25	0.25	0.25	0.25
model	form	0.25	0.25	0.25	0.25
model	dao	0.625	0.125	0.125	0.125
model	manager	0.25	0.25	0.25	0.25
model	none	0.943	0.011	0.034	0.011
view	controller	0.25	0.25	0.25	0.25
view	service	0.25	0.25	0.25	0.25
view	validator	0.25	0.25	0.25	0.25
view	context	0.25	0.25	0.25	0.25
view	servlet	0.25	0.25	0.25	0.25
view	web	0.25	0.25	0.25	0.25
view	aspect	0.25	0.25	0.25	0.25
view	form	0.7	0.1	0.1	0.1
view	dao	0.25	0.25	0.25	0.25
view	manager	0.25	0.25	0.25	0.25
view	none	0.172	0.766	0.016	0.047
controller	controller	0.786	0.071	0.071	0.071
controller	service	0.583	0.083	0.25	0.083
controller	validator	0.812	0.062	0.062	0.062
controller	context	0.125	0.125	0.625	0.125
controller	servlet	0.083	0.083	0.75	0.083
controller	web	0.071	0.071	0.786	0.071
controller	aspect	0.7	0.1	0.1	0.1
controller	form	0.25	0.25	0.25	0.25
controller	dao	0.25	0.25	0.25	0.25
controller	manager	0.625	0.125	0.125	0.125
controller	none	0.312	0.062	0.562	0.062

Figure 5. Probability Distribution Table for the *Type* variable

Layer	Suffix	springmvc	aspectj	hibernate	jdbc	none
model	controller	0.2	0.2	0.2	0.2	0.2
model	service	0.2	0.2	0.2	0.2	0.2
model	validator	0.2	0.2	0.2	0.2	0.2
model	context	0.2	0.2	0.2	0.2	0.2
model	servlet	0.2	0.2	0.2	0.2	0.2
model	web	0.2	0.2	0.2	0.2	0.2
model	aspect	0.2	0.2	0.2	0.2	0.2
model	form	0.2	0.2	0.2	0.2	0.2
model	dao	0.111	0.111	0.111	0.333	0.333
model	manager	0.2	0.2	0.2	0.2	0.2
model	none	0.011	0.011	0.056	0.034	0.888
view	controller	0.2	0.2	0.2	0.2	0.2
view	service	0.2	0.2	0.2	0.2	0.2
view	validator	0.2	0.2	0.2	0.2	0.2
view	context	0.2	0.2	0.2	0.2	0.2
view	servlet	0.2	0.2	0.2	0.2	0.2
view	web	0.2	0.2	0.2	0.2	0.2
view	aspect	0.2	0.2	0.2	0.2	0.2
view	form	0.636	0.091	0.091	0.091	0.091
view	dao	0.2	0.2	0.2	0.2	0.2
view	manager	0.2	0.2	0.2	0.2	0.2
view	none	0.354	0.015	0.015	0.015	0.6
controller	controller	0.733	0.067	0.067	0.067	0.067
controller	service	0.077	0.077	0.077	0.077	0.692
controller	validator	0.059	0.059	0.059	0.059	0.765
controller	context	0.111	0.111	0.111	0.111	0.556
controller	servlet	0.077	0.077	0.077	0.077	0.692
controller	web	0.067	0.067	0.067	0.067	0.733
controller	aspect	0.091	0.636	0.091	0.091	0.091
controller	form	0.2	0.2	0.2	0.2	0.2
controller	dao	0.2	0.2	0.2	0.2	0.2
controller	manager	0.111	0.111	0.111	0.111	0.556
controller	none	0.059	0.059	0.059	0.059	0.765

Figure 6. Probability Distribution Table for the *ExternalAPI* variable

Using the Bayesian network generated by Weka, 110 instances of the ARFF file are corrected classified. That is, we have around 96% 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 the following figure the classification results are shown:

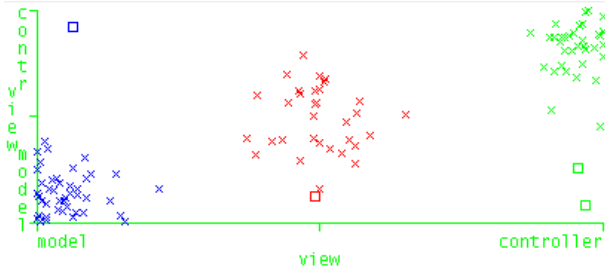


Figure 7. Classification Results. Note that a *Square* represents an Error and a *Cross* represents a Success.

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 for creating the probabilistic model, as explained before.

### B. Relationship analysis

Complementing classification results, a further analysis is performed to guarantee that invalid dependencies, as stated in the Model 2 pattern [4], a variation of MVC for web projects, are not found in the implementation components. This analysis will also verify that the grouping of components by packages and directories is consistent with a common web project distribution.

The Model 2 pattern is depicted in the following Figure.

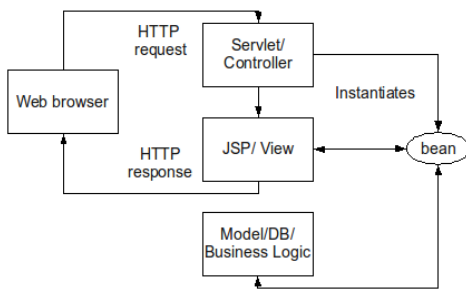


Figure 8. Model 2 Pattern

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.

Once the components are classified, all the code packages 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. For non-code components, a similar logic applies for the directories containing them.

### 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 finally linking the components by their inter-project relationships. The components identified as invalid during the Relationship analysis are highlighted to allow further examination by the development team.

### D. Implementation for Java Web 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 [15]. 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 [14], thus, it is restricted to this environment.

In order to use the Bayesian network, previously generated with Weka, we need to be able to assign for each of the analyzed components, values for the *Type*, *ExternalAPI* and *Suffix* random variables. This has to be 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 [16]. Based on the Visitor pattern [17], 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 as it, 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 performed after 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 dependencies list. For Java classes, the *Type* variable is assigned the value *java*. The *Suffix* variable can be given values directly from the java classes file names.

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. This is done in the *MvcAnalyzer* class using Weka's Java API.

Once the classification process is over, the Relationship analysis is performed for all the project components, in order to identify invalid relations among them. Valid relation definitions are stored in the *Layer* class, which is the component used to identify the grouping layers of the MVC pattern, and as such, instances of this class are associated to the project components according to their classification. 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 Dot library [18]. Layers are identified by a combination of a color and a shape, according to the following Figure:

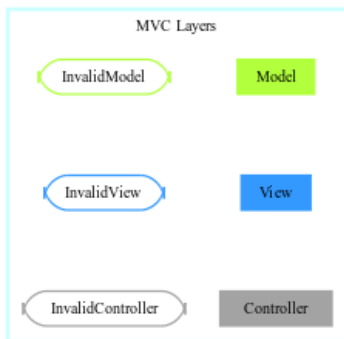


Figure 9. MVC Layers

We can see that each layer has its invalid counterpart, that differs from it in shape and fill style but not in color. We use these styles to depict the project components classification. For instance, a fragment of the view generated for the Distributed Query Optimizer project is shown next:

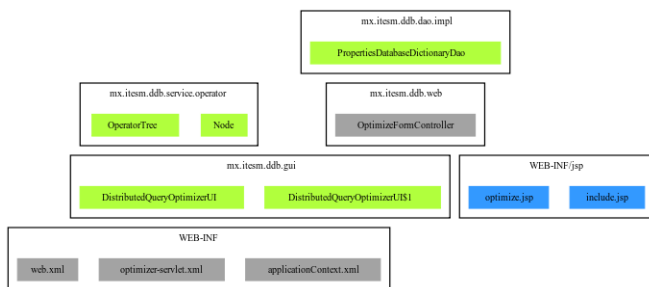


Figure 10. Fragment of an architecture view showing valid components.

Analyzing this fragment of the 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 one or more components. Note that only valid components are shown in the fragment.

To see how invalid components are shown in the view, a fragment of the view generated for the PetClinic application is depicted in the following figure:

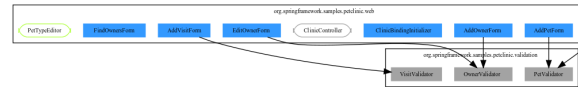


Figure 10. Fragment of an architecture view showing invalid components.

We can see that two components of this package have been marked as invalid. This is because their classification group, *Model* and *Controller* respectively, differ from the package's classification group, *View*, being that in this package most of the components were classified as *View*.

To show how a complete architecture view is presented to the user, the architecture view generated for the PetClinic application is depicted in the following Figure:



Figure 11. Complete architecture view

The implementation details explained so far are part of a generic module. Besides this module, a plugin for the Eclipse platform was developed. This plugin [15] allows us to invoke the MVC classifier process from within an eclipse view, and see the results within another eclipse view. The plugin is designed to work with Project elements and WAR files, by adding a context menu entry when these objects are selected by a right-click in one of eclipse's views.

For Project elements, the menu displayed to the user is: “Pattern Views → Generate MVC View for Project files”

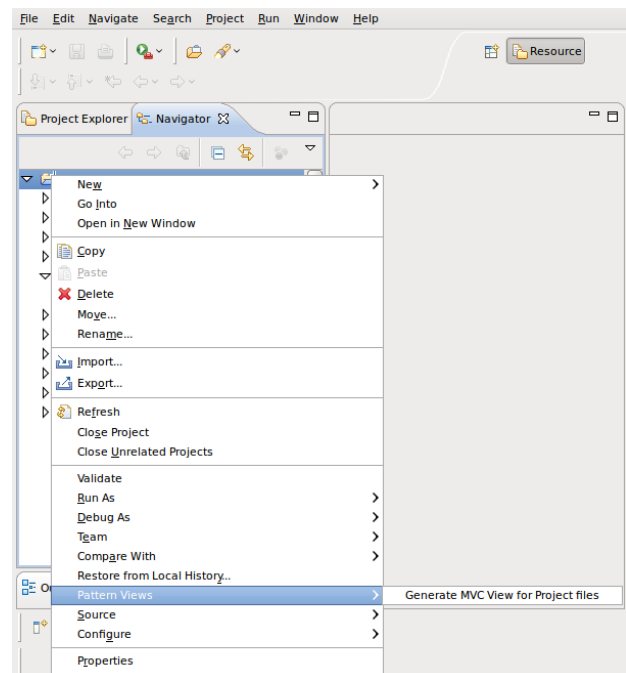


Figure 12. Context Menu for Project elements

For WAR files, the menu displayed to the user is: “*Pattern Views* → *Generate MVC View for WAR file*”:

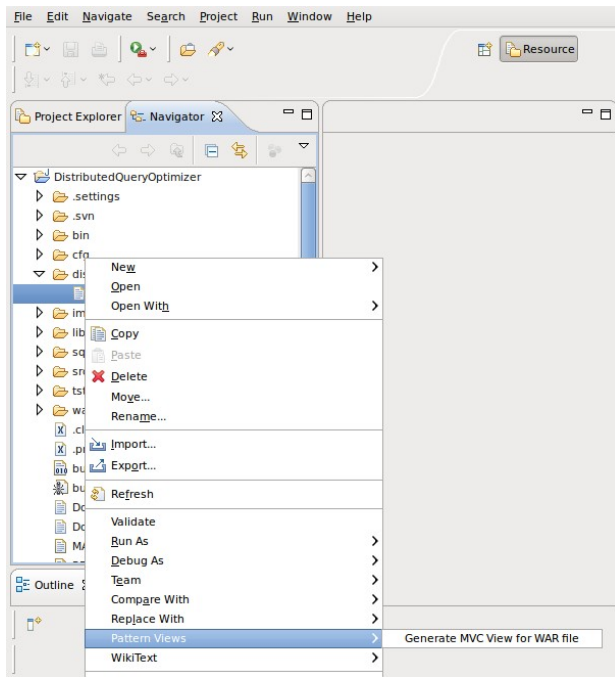


Figure 13. Context Menu for WAR files

If any of the context entries is selected, an eclipse view, *MvcView*, is opened to show the generated architecture view, using the SVG Salamander library [19].

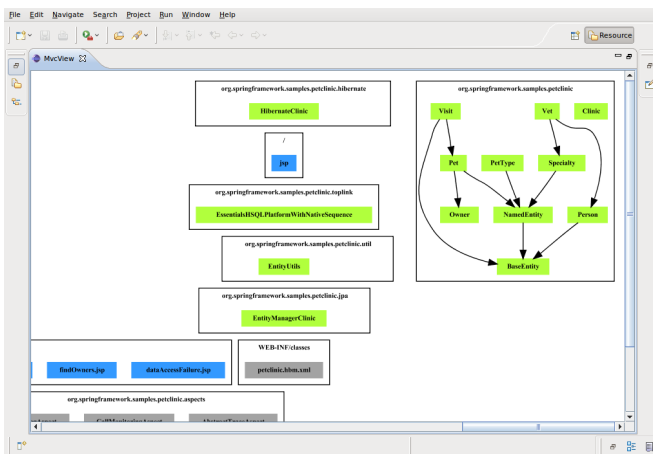


Figure 14. MvcView

## V. CONCLUSIONS

In this paper, I proposed an approach for the generation of an architecture view for a MVC Web application, by using a bayesian network classifier over the components that comprise the application, and identifying invalid relationships among them. The bayesian network is built by using the Weka software, and trained with data of a small set of representative Java applications. The effectiveness of the generated probabilistic model is promising.

To test the 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, we are able to use the classifier with any Java web project and WAR file, generating the architecture view into a SVG file.

## VI. 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 more Java technologies.

## REFERENCES

- [1] Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice*, Second Edition (2 ed.). New York: Addison-wesley Professional.
- [2] Gorton, I. (2006). *Essential Software Architecture* (1 ed.). New York: Springer.
- [3] Java BluePrints - J2EE Patterns. (2000, January 1). Sun Developer Network. Retrieved December 6, 2009, from <http://java.sun.com/blueprints/patterns/MVC-detailed.html>
- [4] *Designing Enterprise Applications with the J2EE Platform*, Second Edition. (2002, January 1). Sun Developer Network. Retrieved December 6, 2009, from [http://java.sun.com/blueprints/guidelines/designing\\_enterprise\\_applications\\_2e/web-tier/web-tier5.html](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html)
- [5] Neapolitan, R. E. (2003). *Learning Bayesian Networks* (1st ed.). Alexandria, VA: Prentice Hall.
- [6] Jensen, F. V., & Nielsen, T. (2007). *Bayesian Networks and Decision Graphs* (Information Science and Statistics) (2 ed.). New York: Springer.
- [7] Ziv, H., Richardson, D., & Klösch, R. (n.d.). *The Uncertainty Principle in Software Engineering*. Object Technology Jeff Sutherland. Retrieved December 6, 2009, from <http://www.jeffsutherland.org/papers/zivchaos.html>
- [8] Maqbool, O., & Babri, H. (2007, April 11). *Bayesian Learning for Software Architecture Recovery*. IEEE Xplore. Retrieved December 6, 2009, from [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=4287309](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4287309)
- [9] Lungu, M. (2008, October 1). *Software Architecture Recovery: The 5 Questions You Always Asked Yourself About*. SlideShare. Retrieved December 6, 2009, from <http://www.slideshare.net/mircea.lungu/software-architecture-recovery-in-five-questions-presentation>
- [10] Johnson, R., & Hoeller, J. (2008, November 8). Chapter 13. *Web MVC framework*. SpringSource.org. Retrieved December 6, 2009, from <http://static.springframework.org/spring/docs/2.5.6/reference/mvc.html>
- [11] Hedayet, M. (2007, November 27). *Spring MVC Tutorial*. Himu's Attempt at Blogging. Retrieved December 6, 2009, from <http://mhimu.wordpress.com/2007/11/27/spring-mvc-tutorial/>
- [12] Castrejón, J. (2009, October 22). *ddb-query-optimizer*. Google Code. Retrieved December 6, 2009, from <http://code.google.com/p/ddb-query-optimizer/>
- [13] Weka 3 - Data Mining with Open Source Machine Learning Software in Java. (2009, June 4). Computer Science Department, University of Waikato. Retrieved December 6, 2009, from [http://www.cs.waikato.ac.nz/~ml/weka/index\\_documentation.html](http://www.cs.waikato.ac.nz/~ml/weka/index_documentation.html)
- [14] Eclipse Project. (2009, September 17). Eclipse.org. Retrieved December 6, 2009, from <http://www.eclipse.org/>
- [15] Castrejón, J. (2009, October 26). *mvc-analyzer*. Google Code. Retrieved December 6, 2009, from <http://code.google.com/p/mvc-analyzer/>
- [16] ASM. (2009, June 11). OW2 Consortium. Retrieved December 6, 2009, from <http://asm.ow2.org/>
- [17] Visitor Pattern. (n.d.). *Design Patterns - Object Oriented Design*. Retrieved December 6, 2009, from <http://www.oodeign.com/visitor-pattern.html>
- [18] Bilgin, A., Ellson, J., Gansner, E., Hu, Y., Koren, Y., & North, S. (2009, June 16). *Graphviz - Graph Visualization Software*. Graphviz. Retrieved December 6, 2009, from <http://www.graphviz.org/>
- [19] McKay, M. (n.d.). *SVG Salamander*. java.net. Retrieved December 6, 2009, from <https://svgsalamander.dev.java.net/>