Driving the refactoring of Java Enterprise Applications by evaluating the distance between application elements

Fabrizio Perin

Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

***Abstract*—Java Enterprise Applications (JEAs) are complex systems composed using various technologies that in turn rely on languages other than Java, such as XML or SQL. Given the complexity of these applications, the need to reverse engineer them in order to support further development becomes critical. In this paper we show how it is possible to split a system into layers and how is possible to interpret the distance between application elements in order to support the refactoring of JEAs. The purpose of this paper is to explore ways to provide suggestions about the refactoring operations to perform on the code by evaluating the distance between layers and elements belonging those layers. We split JEAs into layers by considering the kinds and the purposes of the elements composing the application. We measure distance between elements by using the notion of the shortest path in a graph. Also we present how to enrich the interpretation of the distance value with enterprise pattern detection in order to refine the suggestion about modifications to perform on the code.**

***Keywords*-Reverse engineering; Java Enterprise; Architecture.**

I. Introduction

Since Java 2 Platform Enterprise Edition (J2EE) was introduced in 1999 it has become one of the standard technologies for enterprise application development. J2EE applications are complex systems composed using various technologies that in turn rely on languages other than Java, such as XML or SQL. In order to perform different kinds of analyses on JEAs it is important to collect information from all sources and put it together in a model that can include everything relevant. In this paper we focus our attention on analyzing the structure of JEAs. As known from the literature [1], [2], [3], [4] JEAs, by their very nature, can be split into layers. Each layer can differ from the others in the type of the elements that it contains or in the task that those elements have been created to perform. Evaluating the distance between different layers and between elements belonging those layers we can reveal violations of the application's architecture that should be modified in order to improve readability and maintainability.

In this paper we propose a technique to drive this refactoring. We describe the layering scheme that we adopt, the index that we use to identify architectural violations and how we can modify the interpretation of that index using enterprise pattern detection. We plan to implement our proposal in Moose [5], a software analysis platform, in order to exploit our existing infrastructure for static and dynamic analysis. Moose is a reengineering environment that provide several services including a language independent meta-model. On top of Moose have been build several tools that provide different services like: static analysis, dynamic analysis, software visualization, evolution analysis.

II. Layering

Java Enterprise Applications are complex software systems composed of different elements. Because those elements have different purposes and behaviors JEAs can be split into layers [1]. Comparing different types of layering schemes [1], [2], [3], [4] we split the applications into 4 layers: Presentation, Service, Business and Data layer. Every element belonging to a layer has a specific purpose and they work together to solve the user's problems. The Presentation layer contains all elements concerning the front-end of the application such as a rich UI or an HTML browser interface. The Service layer is part of the Business layer and contains all those elements that define the set of available operations and manage the communications between the Presentation layer and the domain logic classes. The Business layer includes all classes that implement and model the domain logic. The Data layer contains all classes that access the database and map the data into objects that can be used by the application. In Figure 1 illustrates the layering system that we adopt. The large external rectangles represent the layers.

When implementing a service it is always important to create a complete structure that involves all layers. There are two main reasons for this: the first, and most important one, is related to code understanding and the second to maintainability. If, for some reason, a service does not need to process some data, an element belonging to the Presentation layer can invoke directly an element in the Data layer. However in this way whoever will read the code will miss the domain model related to that particular service. So it becomes much more difficult to understand the code.

It is also important go through all layers for reasons of maintainability. If at the beginning of development it is not necessary to process some data, it could became necessary afterward and it is usually tricky to modify the structure of the code preserving understandability.

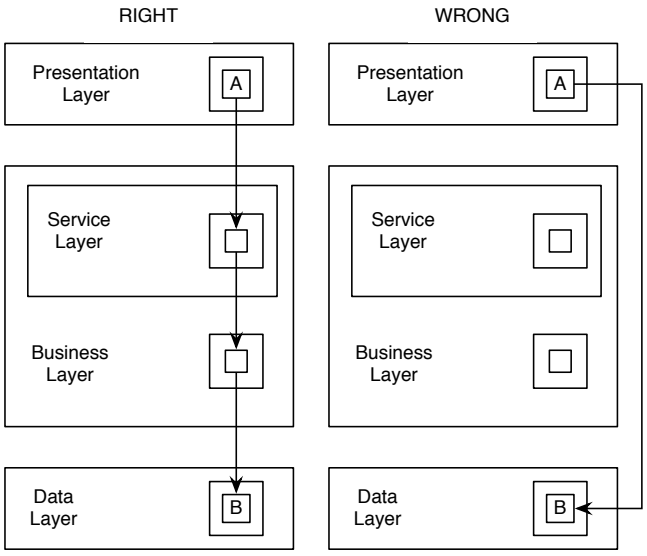

Figure 1. Layering scheme and basic invocation chian.

III. Distance between elements

We use value of the distance between JEA's elements to identify violations in the architecture. We compute the distance between elements using the notion of the shortest path in a graph. We can therefore use graph theory to solve our problem [6]. In the following we summarize the idea of distance used to solve the shortest path problem.

We have a weighted, directed graph $G = \langle V, E \rangle$, with a weight function $\omega : E \rightarrow \mathrm{R}$ mapping edges to real-valued weights. The weight of a path $p = \langle \upsilon_0, \upsilon_1, \upsilon_2, \ldots, \upsilon_n \rangle$ is the sum of all weights of its edges:

$$\omega(p) = \sum_{i=1}^{k} \omega(\upsilon_{i-1}, \upsilon_i) \tag{1}$$

The shortest path weight from u to υ is defined as:

$$\delta(u, \upsilon) = \begin{cases} \min\{\omega(p) : u \rightsquigarrow \upsilon\} & \text{if there is a path} \\ & \text{from } u \text{ and } \upsilon, \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

Every edge in the graph has a weight that is necessary to find which is the shortest path between two elements. In our case we want to assign to each of them the weight 1. It is possible to calculate the minimum number of invocations from one method to another using Dijkstra's algorithm. We define the *distance* between two elements as the shortest path between those two elements.

We will define threshold values to determine which distance is correct and which is not. To simplify the concept in this work, even if the Presentation layer could contain HTML or JS Pages as well as GUI elements in any language, in the following we will consider an element contained in this layer as a class with methods.

The basic distance to calculate is the distance between different methods. The distances between classes and layers are derived from the distance between methods, therefore it is not necessary to apply the algorithm used to calculate distances on those methods, it is just necessary to regroup the methods of a path as part of a class or a layer.

In Figure 1 on the left we show the basic invocation chain that implements a normal user request. This chain contains classes and the smallest squares are methods or a generic JSP or HTML page. In Figure 1 on the right we also show what we consider to be a wrong invocation chain. In fact in this case the element A belonging to the Presentation layer invokes directly the method B belonging to the Data layer.

The distance between classes and layers is important to calculate because we cannot be sure that everything is fine just looking at the distance between methods. If the distance between methods has an acceptable value this doesn't means that every layer is touched in the implementation.

We will present in the following some examples that cover some normal cases that can be found in a normal implementation of an enterprise application. It is important to underline that the real threshold values to adopt to evaluate the code are still to be decided and they will be defined by analyzing some huge industrial case studies with a number of classes up to 1800. Below we exemplify our idea considering as right a distance value between two elements belonging the Presentation and the Data layer equals to 3.

Example 1: In Figure 1 on the left: The distance between method A and method B is 3 as well as the distance between the class that contains A and the class that contains B and the distance between the Presentation layer and the Data layer. We consider this situation a basic right implementation where all layers are touched. If instead we consider two classes, the first belonging to the Presentation layer and the second belongs to the Data layer, then if the distance between those classes is 1 it means that there is a direct invocation from the Presentation layer to the Data layer. This is the most basic and recurrent case of wrong implementation.

Example 2: In Figure 2 are shown a couple of correct paths. On the left side the distance between method A and method B is 4, instead the distance between the class that contains A and the class that contains B as well as the distance between the Presentation layer and the Data layer is 3. From those

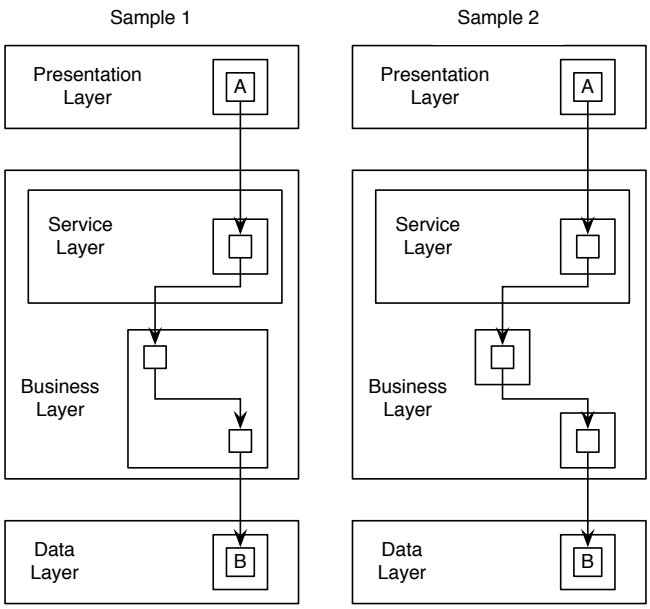

Figure 2. Layering scheme and basic invocation chian.

values we can see that all layers have been touched and that in a class there is an internal invocation because of the difference between the distance between methods and the distance between classes. On the right in Figure 2 all distance values are the same and equal to 4. Also in this case we can see that everything seems fine. What is important in a situation like this is that the number of invocations is not too high. A too high value is a symptom of complexity, so maybe that service implementation could be simplified.

Example 3: In Figure 3 on the left the distance between method A and method B is 3 so we consider it right. On the other hand the distance between their classes is just 2 as well as the distance between the Presentation layer and the Data layer. What is missing is a representation of the domain logic. On the right in Figure 3 the distance between A and B and the distance between the classes that contain them is 4 so it is right but the distance between layers is 2. What is missing is an entry point for that specific service because method A accesses directly a method belonging to the Business layer.

IV. Distance and Enterprise Patterns

There is a large body of development patterns gathered by the engineering community. There are patterns for enterprise applications [1] in general and patterns for J2EE [4] in particular. The description of design patterns provides information about the structure, the participant's roles, the interaction between participants and, above all, the intent for which they should be used. Our intent is to mix the value of the distance between elements and data source architectural patterns [1] tuning the results obtained by just looking for the distance index.

By being able to identify data source architectural patterns in the applications it is possible to provide more specific suggestions on the operation to accomplish during the refactoring. It is also possible to identify potential errors in a correct invocation sequence or vice versa.

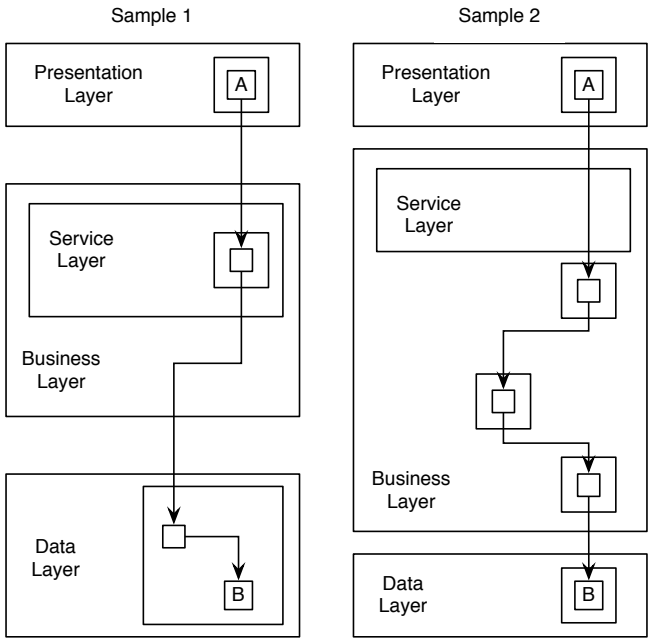

Figure 3. Layering scheme and basic invocation chian.

For example in Figure 1 on the right is shown a wrong invocation where the distance between A and B is 1. In this case the business logic is missing as well as the invocation to the Service layer that define the services available in the system. A standard modification to the code will be to implement one class belonging to the Business layer and another belonging to the Service layer. Supposing now that the class that contains B implements the pattern Active Record [1]. By it own definition, this pattern should contain some domain logic. So the modification to enact is not just to implement the missing classes in the Business and the Data layer but also to move the domain logic in the new class down to the Business layer.

V. Related Works

Some effort has been already spent in the context of architectural conformance checking [7], [8], [9]. In particular [9] extracting information from source code and byte code in Java and C++ and storing this information in a database that models all information that can be extracted from the code. From this information it is possible to perform different kind of analyses like checking illegal relationships within layers.

There are some differences between our work and [9]. We plan to apply our idea to an enterprise system that contains not only Java but also other languages such as XML, JS or SQL. The information harvested from the system will be modeled with FAMIX [10]. Our model will include all aspects concerning an enterprise application: the FAMIX meta model will not only contain structural information but also higher level information such as methods involved in a transaction. Using Mondrian [11] on the information contained in the meta model we can generate many different software visualizations on the code. Our intent is not only to identify which part of code could contain errors or inconsistencies but also to suggest possible modifications for the refactoring.

VI. Conclusion

In this paper we summarized our proposal to drive the refactoring of JEAs by comparing the distance between the application's elements to that of the data source architectural patterns.

In particular we presented the layering scheme that we adopt to regroup different elements of a JEAs. We also explore how to apply the concept of method distance to create an index that can be used to detect the presence of a wrong application structure. Finally we relate the concept of distance to the data source architectural patterns in order to modify the interpretation of the distances. In this case we propose to compile a catalogue of patterns and distances together with heuristics to drive the refactoring.

The basic idea is that every service has to be implemented touching every layer starting from the Presentation one. The catalog will be able to indicate what is wrong in the implementation and how the code should be modified in order to have a right structure. The approach could also be used to expose code that is too complex, *i.e.*, if the distance is too high.

We plan to implement our proposal in FAMIX [10] which already includes a generic meta-model for Object-Oriented application that can be extended to analyze Enterprise applications in Java. We want to refine FAMIX by adding all parts and relations that are necessary to model a JEA. Based on a consistent meta-model, it is possible to define a quality model based on metrics and pattern detection. We will evaluate the performance impact of calculating every time the distance or cacheing it. Another solution could be pre-compute all distances between all elements. In this case the Floyd-Warshall algorithm will fit better.

In order to validate this work we plan to perform experiments using an industrial partner we have been collaborating with.

Acknowledgments We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (Project no. 2234, Oct. 2007 – Sept. 2010). We would also want to thank Tudor Gîrba, Oscar Nierstrasz for their comments on this paper and their support on this project.

References

[1] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2005.

[2] F. Marinescu, *Ejb Design Patterns: Advanced Patterns, Processes, and Idioms with Poster*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[3] K. Brown and G. C. et al., *Enterprise Java Programming with IBM Websphere*. Addison Wesley, 2001.

[4] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001.

[5] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. New York NY: ACM Press, 2005, pp. 1–10, invited paper. [Online]. Available: http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf

[6] T. H. Corman, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[7] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 1995, pp. 18–28.

[8] B. Laguë, C. Leduc, A. L. Bon, E. Merlo, and M. Dagenais, "An analysis framework for understanding layered software architectures," in *Proceedings IWPC '98*, 1998.

[9] W. Bischofberger, J. Kühl, and S. Löffler, "Sotograph – a pragmatic approach to source code architecture conformance checking," in *Software Architecture*, ser. LNCS. Springer-Verlag, 2004, vol. 3047, pp. 1–9.

[10] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*. IEEE Computer Society Press, 2000, pp. 157–167. [Online]. Available: http://scg.unibe.ch/archive/papers/Tich00bRefactoringMetamodel.pdf

[11] M. Meyer, T. Gîrba, and M. Lungu, "Mondrian: An agile visualization framework," in *ACM Symposium on Software Visualization (SoftVis'06)*. New York, NY, USA: ACM Press, 2006, pp. 135–144. [Online]. Available: http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf