Recovery and Analysis of Transaction Scope from Scattered Information in Java Enterprise Applications

Fabrizio Perin, Tudor Gîrba, Oscar Nierstrasz

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

***Abstract*—Java Enterprise Applications (JEAs) are large systems that integrate multiple technologies and programming languages. Transactions in JEAs simplify the development of code that deals with failure recovery and multi-user coordination by guaranteeing atomicity of sets of operations. The heterogeneous nature of JEAs, however, can obfuscate conceptual errors in the application code, and in particular can hide incorrect declarations of transaction scope. In this paper we present a technique to expose and analyze the application transaction scope in JEAs by merging and analyzing information from multiple sources. We also present several novel visualizations that aid in the analysis of transaction scope by highlighting anomalies in the specification of transactions and violations of architectural constraints. We have validated our approach on two versions of a large commercial case study.**

Keywords: Reverse engineering, Java, Enterprise Application, Transactions, Visualization.

I. Introduction

An increasing number of companies use complex Java Enterprise Applications (JEAs) to solve their business problems. JEAs are complex because they are composed of multiple parts written in different languages and they are distributed. In this context information concerning the application's elements and their relations is spread over various sources. This causes the developers to lose the overview of the system and hides inconsistencies in the code. Information can potentially reside in XML configuration files (to define Enterprise Java Beans (EJBs) in the version 2.1 [1] or used by the build system), source code (*e.g.*, Java code or JS Pages), SQL scripts, and so on. The heterogenous nature of JEAs can make it difficult to analyze and verify desirable properties and architectural constraints.

In this paper we tackle the problem of identifying the transaction scope of applications. Application transactions have been defined in JEA to guarantee atomicity of operations and to provide a certain level of isolation of services accessed by different clients. Transactions aid the application programmer to provide the desired level or reliability and consistency. On the one hand it is important to ensure that critical services are properly contained within a transaction scope, while on the other hand starting unnecessary transactions should be avoided for performance reasons. We define a transaction as unnecessary if its scope is nested within another transaction scope. In this paper we consequently focus our attention on three specific questions:

1) Which methods are involved in a transaction scope and how are they dispersed?
2) Which methods start unnecessary transactions?
3) Which methods access the database from outside a transaction?

In Java 2 Enterprise Edition (J2EE), a method can be defined as part of a transaction by setting certain attributes in the EJB deployment descriptor of the application. The EJB container takes these attributes into account to manage application's transactions. J2EE specifications defines various transaction propagation semantics. The most common attribute is *required* which creates a new transaction if none is active, or reuse the existing one. A method without an explicit transaction attribute can still be part of a transaction if it is has been invoked by another transactional method.

The advantage of this declarative definition is that it is not necessary to touch the Java source code to manipulate them. Nevertheless, the decoupling between the actual method and its transaction attribute makes it difficult to identify the transaction scope the method lies in. Developers cannot easily verify if a method is transactional or not. Furthermore, a transaction can either be started manually using the Java Transaction API or it can be started automatically in applications by Container-Managed Transactions. As a consequence, the usage of transactions is not uniform, further complicating their understanding.

In this paper we present a technique to identify methods and classes involved in transactions, we present an initial meta-model for JEAs that enables their analysis, and we propose three visualizations to highlight anomalies in the definition of transaction scope. We applied our approach to two different versions of an industrial case study consisting of over 1500 Java classes. The technique to identify the transaction scope has been validated by manual inspection mainly using Eclipse. The manual inspection confirmed a high level of precision in the identification of all EJBs and other elements shown in the visualizations. These results have been presented to the company that provided us with the case study, and their feedback has been highly positive.

The paper is structured as follows: section II presents the meta-model that we use to analyze JEAs and the technique we propose to identify transaction scope and other related details. In section III we present the visualizations that expose anomalies in the definition of transaction scope. In section IV we highlight the differences between the two versions of the case study analyzed. In section V we relate our approach to previous work. Finally in section VI we summarize our results, we describe the future work and we conclude.

II. Model and Technique

In this section we present our meta-model for JEAs which enhances the core of the FAMIX meta-model [2]. We also define key terminology and we explain how we identify the scope of transactions.

A. Modeling Java Enterprise Applications

FAMIX [2] is a language independent meta-model that describes the static structure of object-oriented software systems. FAMIX forms the core of the *Moose* [3] platform for software analysis. We therefore decided to extend the FAMIX meta-model to accommodate the heterogeneous nature of JEAs. More specifically, we added entities to describe EJBs and we extended certain entities already present. A simplified FAMIX 3.0 with our extensions is shown in Figure 1 (our extensions are shown in bold).

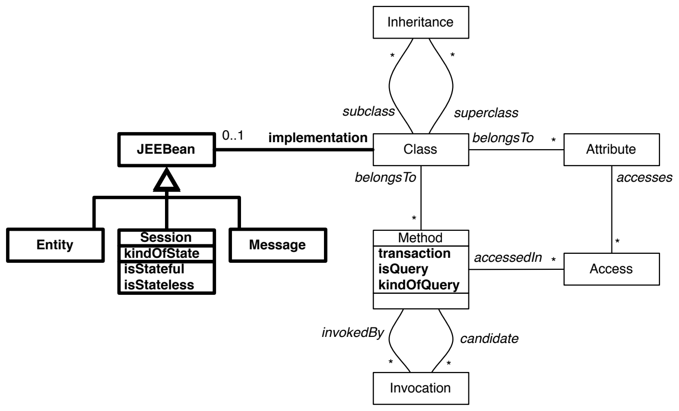

Fig. 1. Enriched FAMIX Meta-Model (extensions are shown in bold)

The new hierarchy on the left represents EJBs. A generic JEE Bean is related to the class that implements it. *Session* specializes *JEE Bean* by adding an attribute to record if the Session Bean is stateful or not. Session Beans are used as an entry point to an application's services and to manage database accesses. Each method in a JEE Bean can be defined as being part of a transaction. We therefore extend the *Method* entity with a transaction attribute. Depending on the value of this attribute, the *EJB Container* [4] will automatically manage transactions. For this reason it is important to specify correctly EJBs and their properties. The transaction attribute is defined only at the method level. It is also possible to specify that all the methods of a bean have the same attribute.

A method may also perform a query using the *java.sql* package. A method can perform a select on the database using the method *executeQuery* and it can modify the database using the method *executeUpdate*. The attribute *isQuery* records whether the method queries the database. The method *kindOfQuery* returns which kind of query it performs. In this work we do not consider tools to manage persistency such as Hibernate but we focus instead on direct queries.

To build an actual model, we use inFusion (a newer version of iPlasma [5]) to parse Java files, and then we separately parse the XML deployment descriptors to associate methods with the corresponding transaction attributes.

B. Identifying Transaction Methods

Having a suitable meta-model we can tackle the problem of identifying transaction scope. We adopt two procedures to answer respectively the initial questions: (1) Which methods are involved in a transaction scope and how are they dispersed? (2) Which methods start unnecessary transactions? (3) Which methods access the database from outside a transaction?

Application transactions in Java guarantee isolation among services and group multiple operations in a unique unit of work. It is not trivial to identify methods involved in a transaction because this is a property with both dynamic and static aspects. A method may be transactional either because it is specified in the deployment descriptor or because it overrides or is invoked by a method that is part of a transaction. We consider the following methods to be part of a transaction[4]:

1) Methods that start a transaction (*i.e.*, their transaction attribute is 'Required' or 'RequiresNew').
2) Methods that override methods that start a transaction.
3) Methods with a transaction attribute 'Mandatory', 'Required' or 'Supports' that are invoked by methods already being part of a transaction.
4) Methods without a transaction attribute that are invoked by methods already part of a transaction.

To expose the transaction scope of methods we proceed in the following way: We define a set of all methods that have a transaction attribute defined. So the initial set includes methods that start a transaction, those that override a method that starts a transaction, and methods that support transactions (a method *supports* a transaction if its transaction attribute is not defined or it is different from 'never' and 'notSupport' [4]). Then we traverse breadth-first the invocation tree of methods invoked by this set. Using this technique we are able to identify all methods that belong to a transaction scope. Before explaining how we identify the invocation paths that are encapsulated in a transaction scope we need some definitions:

- An **entry point method** is a method that is not invoked by other methods.
- A **safe path** is an invocation chain that starts from an entry point method involved in a transaction.

- An **unsafe path** is an invocation chain that starts from an entry point that does not start a transaction.

The following formula describes how we identify an unsafe path:

$$\alpha(Q) \quad \cap \quad \omega(\alpha(Q) \cap E) \tag{1}$$

where:

$$\begin{aligned} M &= \text{all methods} && (2)\\ I &\subseteq M \times M \text{ are invocations} && (3)\\ E &\subseteq M \text{ are entry points} && (4)\\ T &\subseteq M \text{ start a transaction} && (5)\\ Q &\subseteq M \text{ perform a query} && (6)\\ \omega(x) &= \{y|(x,y) \in I^*\} && (7)\\ \alpha(y) &= \{x|(x,y) \in I^* \text{ and } x \notin T\} && (8) \end{aligned}$$

M is the set containing all methods of the system. I represents the invocation relation between methods. E, T and Q are respectively the subset of methods that are entry points, the subset of methods that start a transaction and the subset of methods that perform a query. I^* is the transitive closure. Finally, $\alpha(y)$ returns all methods x *preceding* y in some invocation chain, and $\omega(x)$ returns all methods y *following* x in some invocation chain.

Figure 2 illustrates the steps of an algorithm to compute Formula 1 and compute *unsafe paths*. Step 1 consists in the evaluation of $\alpha(Q)$. In accordance with its definition, the result of $\alpha(Q)$ is the set of methods that are part of the invocation chain ending with a method that executes a query and starting with a method that does not start a transaction. Moreover, none of the methods returned by this function start a transaction. Step 2 is to apply ω to the intersection $\alpha(Q) \cap E$. The result of $\omega(\alpha(Q) \cap E)$ is the subset of methods contained in $\alpha(Q) \cap E$ that are part of the invocation chain starting from an entry point that does not start a transaction. Step 3 consists in the evaluation of the intersection of $\alpha(Q)$ and $\omega(\alpha(Q) \cap E)$ in order to clean up the set from methods not strictly related with the unsafe path.

The result is a set containing the methods of all unsafe paths, namely invocation paths starting from an entry point that do not start a transaction and ending with a method accessing the database. In the diagram, the black path is unsafe since it ultimately performs a query without ever starting a transaction.

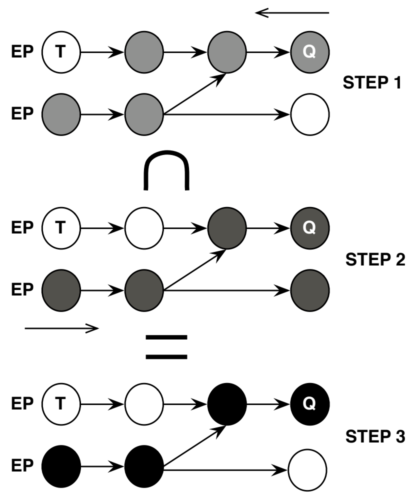

Fig. 2. Unsafe path identification

III. Visualizations

To inspect a JEA from the point of view of application transactions, we devise three interactive visualizations:

- *Transaction flow* provides an overview of the methods involved in transactions (Figure 3),
- *Server Layers* offers an overview of the typical layers in a JEA and helps to identify the misplaced transaction specifications (Figure 4), and
- *Unsafe queries* reveals the methods that query the database without being covered by a transaction (Figure 5).

These visualizations have been developed using the Mondrian visualization engine [6].

In this section we explain in detail each of the three visualizations. We furthermore evaluate the effectiveness of each visualization to analyze an industrial content management system (CMS) to manage customer data. We have analyzed two versions of the same case study released a year apart. The older version is composed of 1938 classes and 55 EJBs, while the newer version is composed of 1527 classes and 43 EJBs. Figures 3, 4 and 5 show elements of the newer version of the case study.

A. Transaction flow

Figure 3 shows all classes and their methods involved in a transaction according to the criteria explained in subsection II-B. Classes and methods are presented as hierarchies that express invocation order. This means that methods of classes on top invoke methods of classes below them. Invocations among classes are represented by gray edges. Internal invocations among methods of the same class instead are organized as a hierarchy going from left to right. The colors of the elements in Figure 3 have the following meaning:

1) *Blue methods* start a transaction. (1) and (7) in Figure 3.
2) *Cyan methods* have a transaction attribute equal to 'Mandatory', 'Required' or 'Supports' and are invoked by methods involved in a transaction. (2) in Figure 3.
3) *Magenta methods* have a transaction attribute equal to 'RequiresNew' and are invoked by methods that start a transaction. (5) in Figure 3.
4) *Grey methods* have no transaction attribute and are invoked by methods already part of a transaction. (3) and (4) in Figure 3.
5) *Orange methods* are entry point methods that have a transaction attribute equal to 'Supports'. (6) in Figure 3.

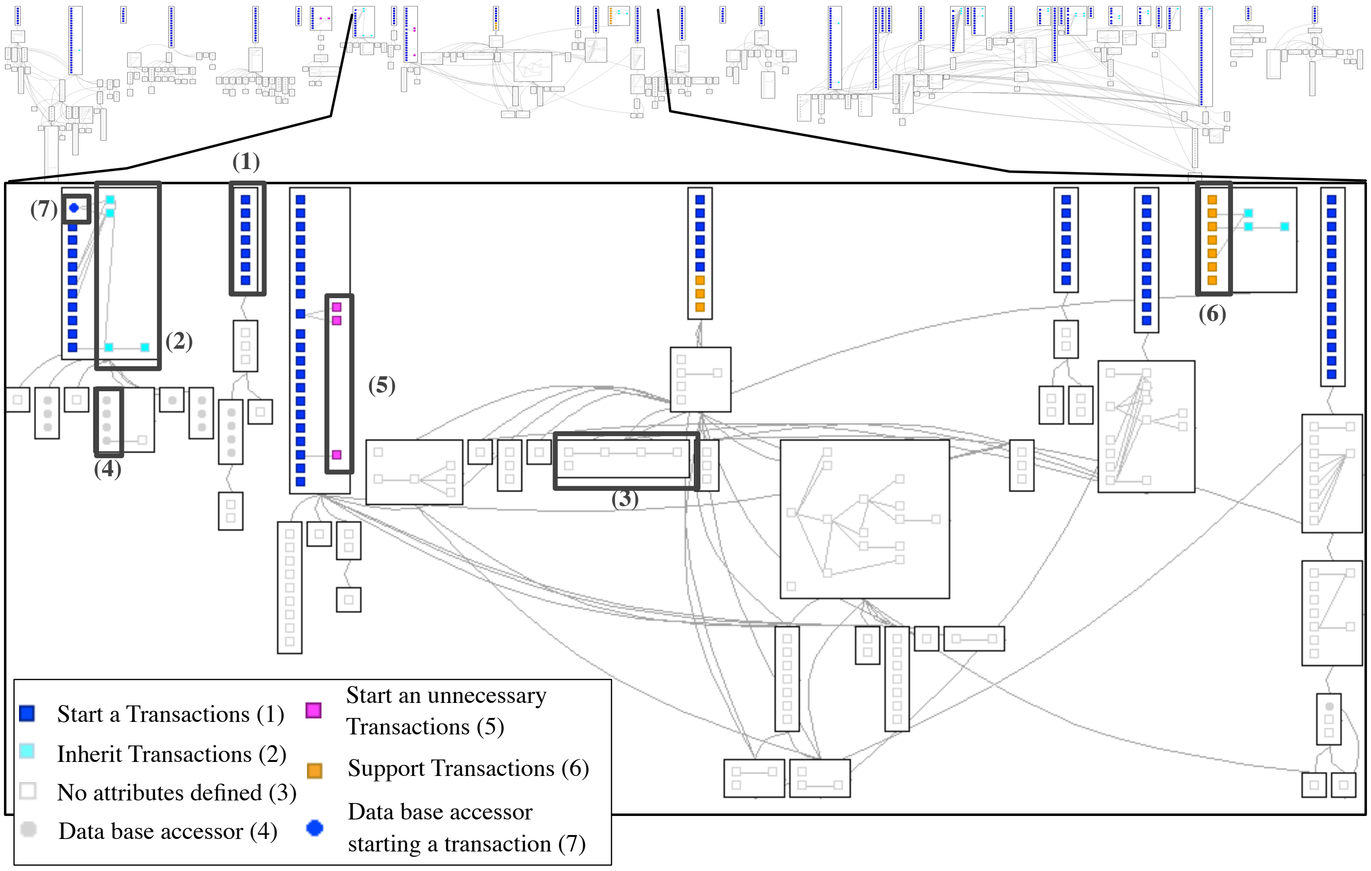

Fig. 3. An excerpt from a Transaction Flow Visualization

The Transaction Flow visualization makes it easy to identify all methods that start an unnecessary transaction (magenta methods, (5) in Figure 3). These methods in fact start a transaction when they could just use the transaction scope of their invoker method. We say that they start an *unnecessary transaction*. Such methods have to be manually checked to verify whether the nested transaction is useless or not. For example a new transaction can be explicitly required to log a message in a database independently of whether the main transaction commits or rolls back.

All hierarchies having no methods starting a transaction at the beginning (blue methods) can lead to problems. These methods support transactions but they do not start one by themselves. This means that either they are within the scope of a transaction started from the application front-end (generally a web interface built using JS Pages but also possibly a GUI written in Java), or the service it uses lies outside a transaction scope. We call such hierarchies *weak paths*. The Transaction Flow visualization is also useful to identify isolated parts of the code that are independent of the rest of the application. In Figure 3 we see two isolated hierarchies at the top left. Hierarchies like these are interesting to identify because they represent services that are “self-contained” in the sense that their entry point and their implementation is not related to other elements of the application. Such hierarchies may also be a sign that opportunities for sharing logic between services have not yet been exploited. On the other hand it is also possible to identify more complex hierarchies with multiple entry points sharing various classes. The identification of these structures may be useful to guide refactoring to make application services more independent. We now evaluate the effectiveness of these visualization on the latest version of an industrial case study.

Case Study: In the case study under analysis all Session beans have methods with a transaction attribute defined and almost all methods start a transaction. We can count 489 methods starting a transaction and 1537 methods involved in a transaction scope. All Session beans appear in the top of the invocation chain, which means that they are actually used as access points for the application services. Using the visualization from Figure 3 we can identify methods that unnecessarily start a transaction. In our case we detected 5 cases.

At the right and in the middle of Figure 3 there are two classes containing methods that support transactions but do not start one. The invocation chains starting from those methods are the only weak paths in the case study. In this case it is necessary to check if the transaction is started in the front-end. If this is the case the EJB container will propagate the

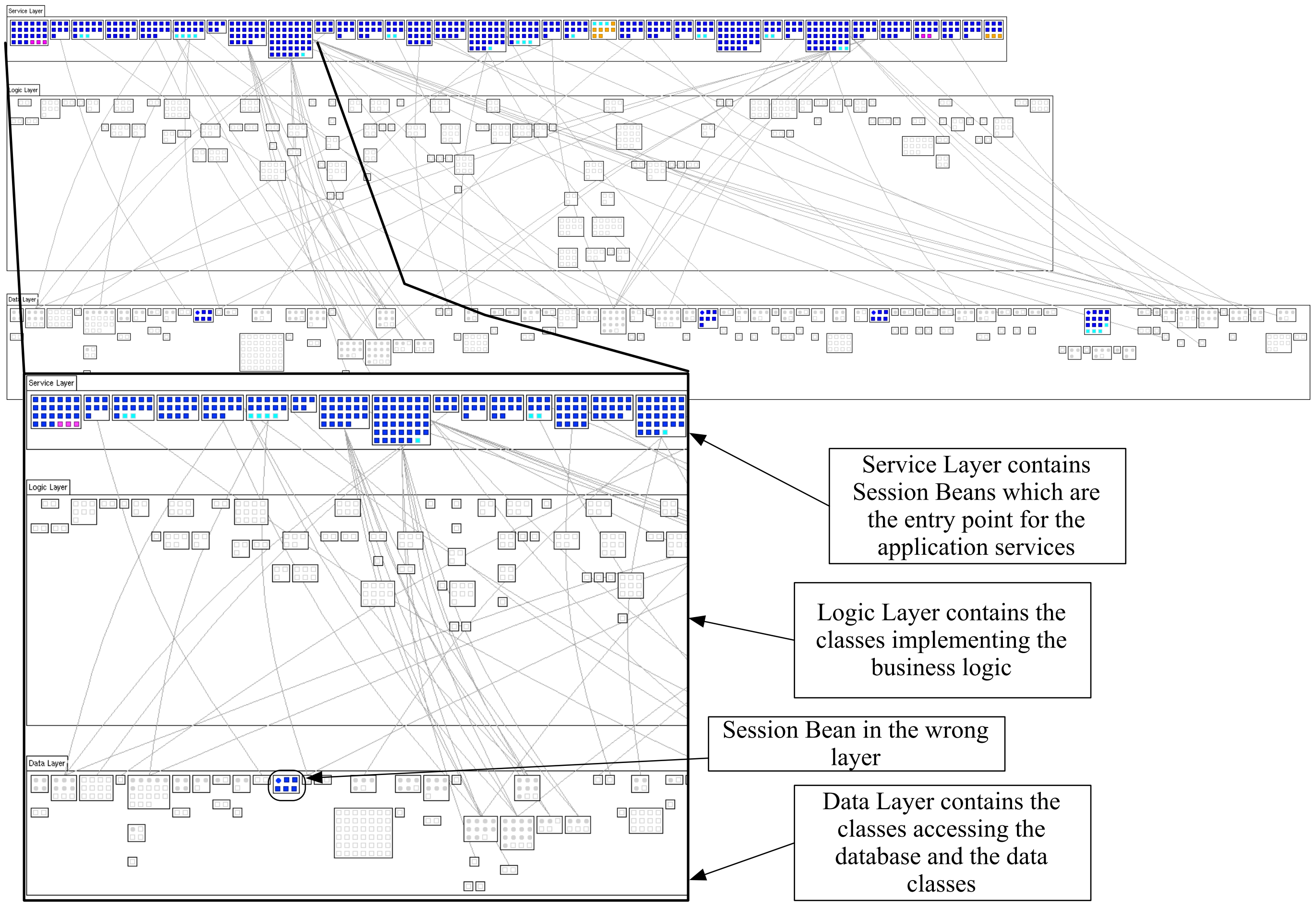
Service Layer
Logic Layer
Data Layer
Service Layer contains Session Beans which are the entry point for the application services
Logic Layer contains the classes implementing the business logic
Session Bean in the wrong layer
Data Layer contains the classes accessing the database and the data classes

Fig. 4. An excerpt from a Server Layers Visualization

transaction scope from the front-end, otherwise these methods are actually operating on the system outside a transaction scope. By showing the entry points that are a potential issue, we ease the investigation of the front-end.

At the left part of Figure 3 there are two small hierarchies starting from a Session Bean without external incoming or outgoing edges. This suggests either that the services that they implement are logically independent from other services, or that the potential for sharing logical functionality with other services has not been exploited. We say that such hierarchies have a “service-oriented” design. In Figure 3 there are also some more complex hierarchies with multiple entry points sharing various classes. In contrast to the “service-oriented” hierarchies at the far left, these more complex hierarchies have multiple entry points that share behavior. We say that such hierarchies have a “use case-oriented” design, since the implementation classes support multiple services.

A final point is the method at the top-left of Figure 3. This method is blue, so it starts a transaction, and it is round, so it accesses the database. This means that the Session Bean it belongs to contains behaviour that is not supposed to be implemented at this level. Ideally the database access should be contained in dedicated classes and not directly in Session Beans. This discovery motivates another visualization, explained in subsection III-B, which is related to the Transaction Flow but has the purpose of identifying architectural violations.

B. Server Layers

Session Beans that start a transaction are used as entry points for services. It is considered good practice to split EA components into presentation, domain and data layers [7]. The domain layer can be further split into a service layer and a domain model. Figure 4 shows the same classes of Figure 3 reorganized into layers. The layer on top is the service layer, in the middle there is the logic layer, and on the bottom there is the data layer. The criteria to split classes into layers are as follows:

1) In the Service layer are visualized all classes implementing a Session bean.
2) In the Data layer we show all classes that (1) are part of the invocation chain starting from the classes belonging to the service layer, and that (2) execute a query, implement the Java interface *Serializable*, or

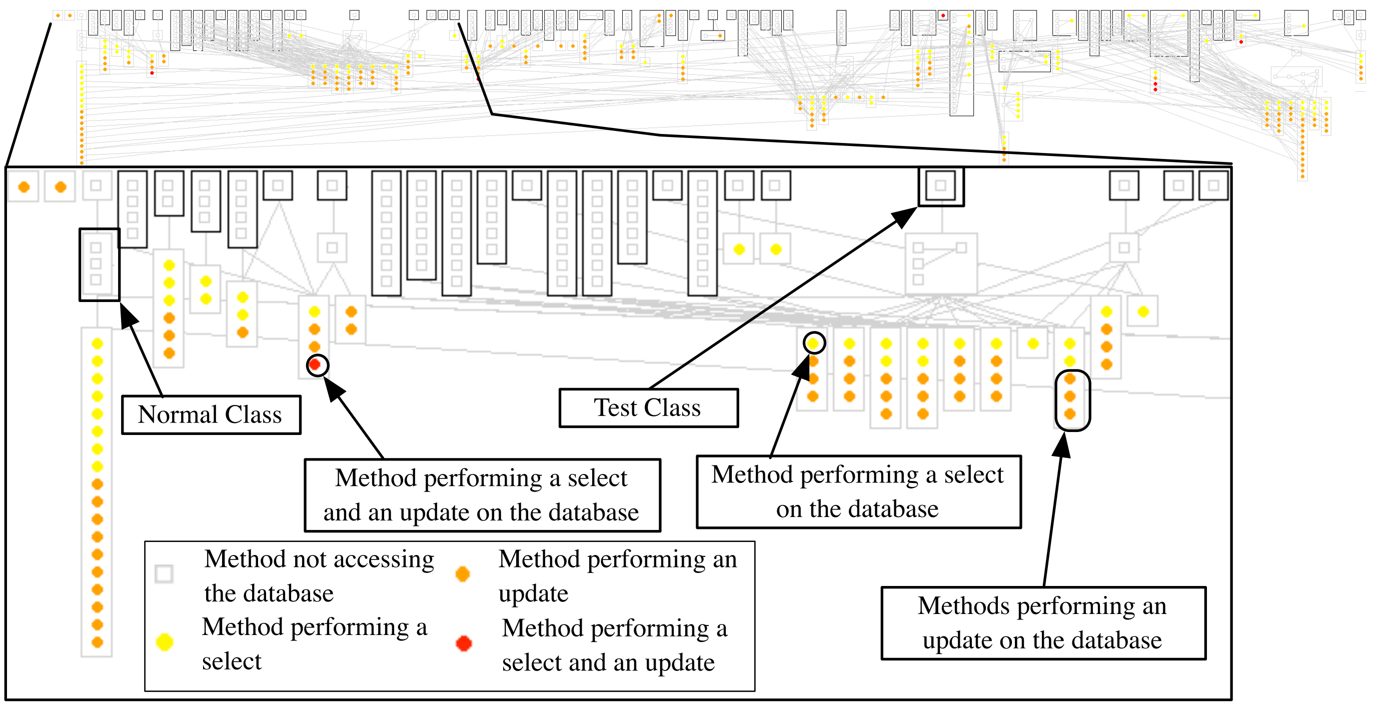
Normal Class
Method performing a select and an update on the database
Test Class
Method performing a select on the database
Methods performing an update on the database
Method not accessing the database
Method performing an update
Method performing a select
Method performing a select and an update

Fig. 5. An excerpt from an Unsafe Queries Visualization

throw an exception from the *java.sql* package. In the case the class implements the interface *Serializable* it can be useful, in order to ensure that the class is actually used to communicate data, to check if the class is also a *data class* [8]. This option will be evaluated in future work.

3) In the Logic layer we show all classes that are part of the invocation chain that started from classes belonging to the service layer, but that are not part of the data layer.

Colors and shapes of objects have the same meaning as in the Transaction Flow visualization. Edges however have a different meaning: the gray edges are invocations that jump a layer by going from the Service layer directly to the Data layer. The purpose of this visualization is to expose structural violations of the elements involved in a transaction. In particular we highlight all Session beans that are in the wrong layer and all invocations that jump a layer. Such violations of architectural constraints not only impact program comprehension, but they may be signs of more serious defects in the application code. The layers and the rules that we implemented can of course fail to match a custom layering structure. This is why in the future we plan to experiment with customization, as in the case of reflexion models [9].

Case study: In Figure 4 there are several edges from the session layer directly to the data layer, suggesting that many services apparently do not have any business logic. Either the business logic has been moved into the Session bean or into the classes belonging to the data layer. In both cases we are dealing with classes that implement behavior external to their competence. We can count 16 Session Beans with gray edges starting from them. Another possibility is that a particular service has no business logic so there are no classes belonging to that layer. This case is also problematic because a service should normally touch all layers — a service without business logic is suspect because it directly exposes the data layer. In any case, software comprehension is compromised because knowledge about the purpose and behavior of a service should appear in the logic layer.

Figure 4 also shows a Session Bean positioned in the wrong layer (in the whole system we identify four of them). These Beans have been classified as belonging to the data layer because they have at least one method that accesses the database. There are two possible interpretations why this is so: either the service is just used to send some data back to the front-end without performing any computation, or the business logic has been pushed to the Session bean instead of creating a separate class belonging to the logic layer. This is similar to the problem related to invocations jumping a layer. In this case we have discovered methods with a behavior outside of their competence.

C. Unsafe Queries

If on the one hand we are able to identify methods involved in a transaction, on the other hand we can show which methods perform queries outside of a transaction. Identifying methods that access the database outside a transaction scope is important to avoid problems of consistency. These methods cannot know if they are working with consistent data or not.

Instead, participating in a transaction ensures that isolation is respected even in the presence of multiple concurrent users of the system. The Unsafe Query visualization shows those hierarchies that end with classes that execute a query to the database and are outside a transaction scope. In particular:

- *black classes* are test classes,
- *grey classes* are not test classes,
- *yellow methods* perform a select,
- *orange methods* perform an update, and
- *red methods* perform both.

The organization of classes is the same as in the Transaction Flow: classes are shown considering the invocation order from the top to the bottom. Edges represent the invocations between methods. Considering this organization, in the top part of the Unsafe Queries visualization are the entry points for unsafe paths. At the bottom, the visualization shows classes that actually perform queries on the database.

Test classes are identified using a naming convention: A class whose name matches the regular expression *".*Test.*"* or a class contained in a package hierarchy whose name matches the regular expression *".*test.*"* is considered to be a test class.

Case study: Considering our case study which is partially shown in Figure 5, we can see that almost all hierarchies start with a test class. This means that during normal execution these paths have to be considered safe. We can count 562 methods outside a transaction scope. In the left part of Figure 5 there is a hierarchy that starts with a gray class. This is an unsafe path that actually accesses the database by reading data without being sure that they are consistent. We count 24 methods belonging to this hierarchy. Also interesting are the two grey classes on the far left of Figure 5. These classes contain a method that performs an update of the database (they might contain other methods that are omitted in the visualizations) and they are not invoked by other methods. These methods pose a risk since they may be invoked directly from the front-end without being part of a transaction scope.

IV. Comparison of Case studies and Validation

In this section we compare the two versions of our case study using the visualizations presented in the previous section to assess the overall quality of the application. In Table I we present some selected metrics related to the systems under analysis to give an idea of the dimension and complexity of the case studies.

Transaction Flow: In Figure 6 we show the Transaction Flow visualizations of both versions. The visualization shows that the application has been strongly modified. The number of single hierarchies without external relations has increased even if there is still a huge hierarchy with many entry points and a large number of classes implementing business logic. In the previous version of the code 10 hierarchies were identifiable, in the new version 15. This difference suggests that in the past the application has been designed and developed considering the point of view of use cases. In the refactoring process many classes have been eliminated and the application adopts a design that is much more service oriented.

	Old	**New**
Classes	1938	1527
Session beans	49	39
Message-driven beans	6	4
Entity beans	0	0
Averave number of methods per bean	14.29	13.15
Methods starting unnecessary transaction	13	5
Methods starting a transaction	622	489
Methods involved in a transaction	2315	1537
Methods involved in an unsafe path	589	562
Isolated call-hierarchies	10	15

TABLE I
Selected case study metrics

Visually, it is also possible to notice that hierarchies are not only cohesive, but also deeper. More classes and methods were involved in the business logic needed to fulfill a service requests. The number of methods starting an unnecessary transaction are reduced from 13 to 5 and the number of beans go down from 49 to 39 which means that the implementation of the services has been modified to eliminate unnecessary transactions. To summarize, all modifications that have been performed on the application improve its structure. The new version of the application is much closer to what we expect from the structure of a JEA: small call-hierarchies with Session beans on top, without relations to other hierarchies and without methods starting unnecessary transactions. Instead all the services are self-contained and transactions are always started if necessary.

Server Layers: Using the Service Layers visualization[1] we identified that the business layer is thinner. This characteristic was visible also in the Transaction Flow visualization but here it is much more evident because the classes are organized into layers. It is more evident that during the refactoring process many classes involved in the computation of requests from services have been removed. Also there are fewer edges. This is maybe because of the lower number of Session Beans but also due to a better organization of the responsibility inside the application.

Unsafe Queries: Applying the Unsafe Queries visualization[2] to the old version of the system we can identify 5 lonely classes that access the database and 5 hierarchies. In total 10 unsafe paths were identified against 3 in the new version. Also in this visualization it can be seen how the refactoring process not only changes the structure but also makes the application's services safer and more efficient.

The case studies were helpful to validate the tool and verify that our technique works correctly not only on small sample code. The validation has been performed manually mainly

[1]The picture is not shown here due to space limitations, but can be accessed at: http://moosetechnology.org/tools/moosejee/casestudy/

[2]Accessible at: http://moosetechnology.org/tools/moosejee/casestudy/

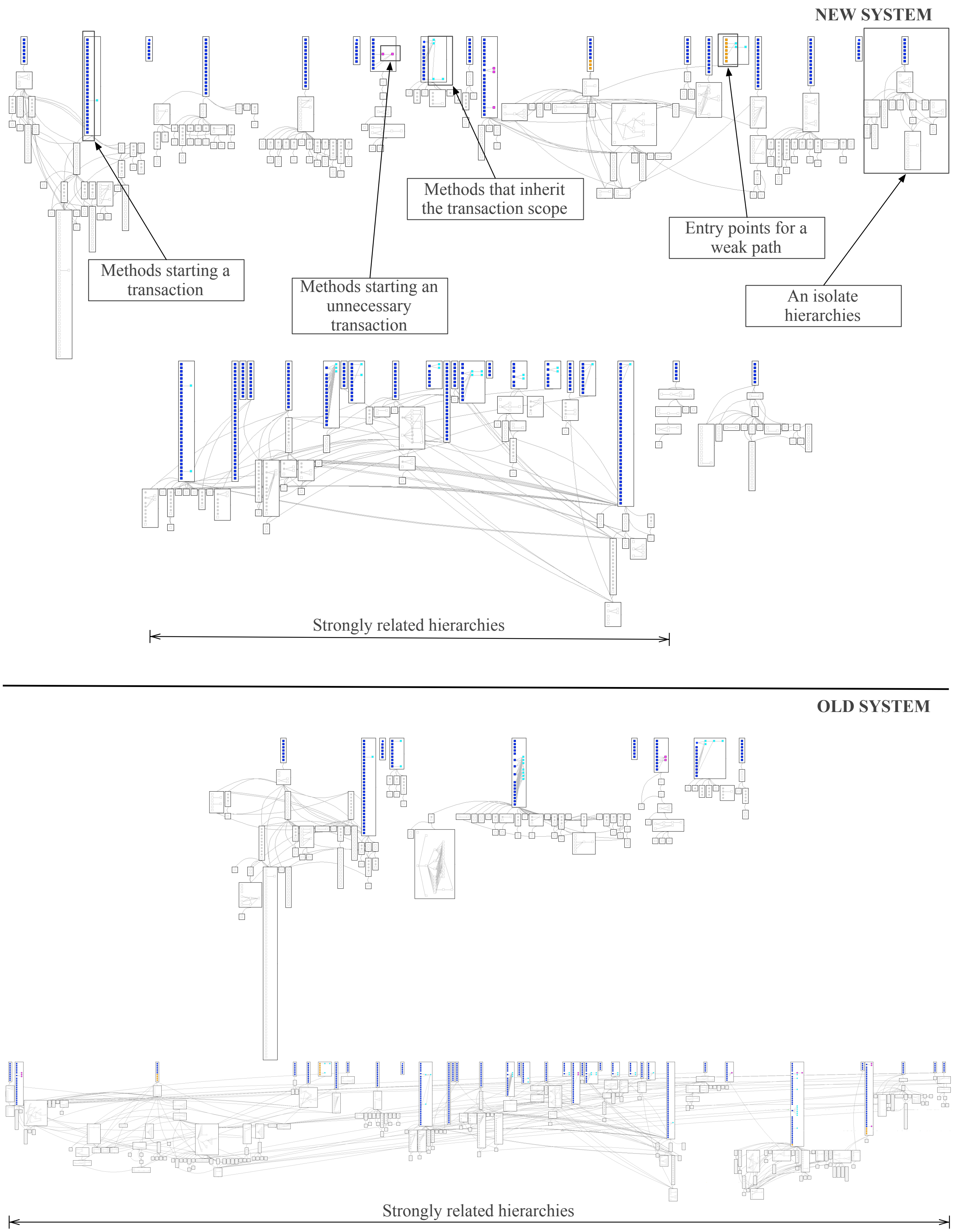
NEW SYSTEM
Methods starting a transaction
Methods starting an unnecessary transaction
Methods that inherit the transaction scope
Entry points for a weak path
An isolate hierarchies
Strongly related hierarchies
OLD SYSTEM
Strongly related hierarchies

Fig. 6. Transaction Flow visualization of two versions of the same case study

using Eclipse. First, we verified that all beans contained in the deployment description have been imported into the model. In both versions we correctly imported all beans. Second, we also manually verified that the transaction attribute is correctly defined for the beans. In this case all methods have been assigned the right transaction attributes. Third, using the Call Hierarchy function of Eclipse we investigated manually a set of random invocation paths in the real case study confirming their correctness.

Considering the Server Layers visualization, all Beans were present in the service layer besides the 4 that access the database. In the Data layer we were able to identify all elements accessing the database using the java.sql package.

Considering the Unsafe Queries visualizations we detected that the methods considered to be entry points for unsafe paths were actually not invoked by other methods.

The manual inspection confirmed a high level of precision in the identification of the elements composing the visualizations. These results have been presented to the company that provided us with the case study, and their feedback has been highly positive. They will inspect all the issues or smells identified using the tool to modify those parts.

V. Related work

Marinescu *et al.* proposed several analyses of Enterprise Applications (EAs). That work focused on design quality assessment [10] based on the identification of roles (*patterns*) for classes and methods, and on the relation between EA source code and databases [11]. The latter work provides a meta-model for relational databases and explains how to relate that schema with a meta-model for EAs. They also investigate the relations between persistent data and the EA proposing an approach to enrich the meaning of foreign keys of database tables by considering the relations between methods. Because EAs are polyglot systems composed using various technologies, it is difficult to describe them with just a generic meta-model. The risk is to miss the description of more specific aspects preventing useful analyses such as the one that we propose. To describe an EA, we need a set of meta-models at a higher level of abstraction. At the moment our meta-model covers different aspects from that proposed by Marinescu.

In software maintenance, system overviews can play an important role in identifying reusable components and assessing modularization [12]. With a completely different kind of visualization and context we create some visualizations with the same purpose: to investigate the presence of visual patterns and to ease the identification of reusable architectural components.

Also other teams have worked on architecture recovery and validation using, for instance, reflexion models [13], [9]. These models are used to recover architecture by capturing developer knowledge and then manually mapping this knowledge to the source code [14], [15]. Through reflexion models it is possible to expose extra information implicit in the code. Our work exposes information implicit in the different technologies used for building Enterprise applications.

Intensional views check conformance of source code to architectural constraints by means of rules expressed in a dedicated logic programming sublanguage [16]. We are unaware of intensional views being used to analyze heterogeneous language projects, or to assess the quality of transactional code.

An interesting study of transactions in EJB has been made at the Vrije Universiteit Brussels [17]. This work highlights the separation of concerns in the transaction management of JEAs. The problem is caused by the scattered definition of the components involved in a transaction, the operations to perform in case of rollback, and the handling of the rollback exception. The work aimed to improve the separation of concerns by using Aspect-Oriented Programming.

Robillard and Murphy proposed a technique to support the analysis of concern propagation in the source code using a Concern Graph [18]. The goal of Concern Graphs was to have a representation of concerns that can be created, manipulated and analyzed while spending minimum effort, and to have a direct mapping to the source code. The Concern Graph has to be manually built from the source code elements through a manual investigation of the code while in our case the procedure is fully automated. Moreover this work is focused on the source code while the basic problem that we have to address is that JEAs are not just Java code. We actually merge information from different sources. In order to address the same result using Concern Graphs the user should dig into XML files to identify which classes have a specific property and then spend time to manually map this information to the Concern Graph.

VI. Conclusion and future work

In JEAs information is spread in many locations (*e.g.*, Java classes, JS Pages, deployment descriptors). By unifying information from multiple sources it is possible to expose problems otherwise difficult to discover. Transactions aid the application programmer with issues like failure recovery and multi-user programming. Because the definitions of methods and their transaction attribute are decoupled in JEAs, it is difficult for developers and designers to identify whether a method is in a transactional scope or not. Information unification is one key point. Another one is to present the result of this unification with visualizations of the whole system code to expose anomalies.

Our contribution consists of a technique to expose the transaction scope of the application's classes and to identify related issues. To reach this objective we extended the FAMIX meta-model to accommodate JEAs. We model EJBs to enable analyses from the perspective of transactions. We developed three novel visualizations to expose structural and behavioral anomalies in the definition and use of transaction scope. To validate the approach, we implemented our complete approach on top of the Moose analysis platform.

The visualizations and the results have been presented to the company that provided us with the case study, and their feedback has been highly positive. The customer checked the anomalies that we identified. They discovered that in the last version of the code all 5 methods starting an unnecessary transaction do so deliberately (for example to obtain a separate transaction scope for logging purposes). They will perform further investigation on the methods that support transactions without starting them and on the strongly related hierarchies for an eventual refactoring. The Session Beans accessing the database directly without passing through all the layers actually perform read operations on the database for visualization purposes unrelated to the business logic, so they decided not to modify them. They also plan a further investigation into the hierarchies outside any transaction scope that actually access the data base to assess eventual problems. They also would like to periodically check their code with these instruments to monitor the status of their application.

Future work will have to take into account other aspects to improve the quality of the presented technique and to address other investigations on JEAs:

The call graphs built from the application analyzed do not use any kind of type approximation. This fact does not affect the techniques and the considerations presented in the paper. Nevertheless using a different and more precise call graph on the same application could lead to different results. In particular the *unsafe query* visualization will have more benefits from a more accurate call graph revealing more precise hierarchies of classes not included in a transaction scope.

In this paper we focused our work on application transactions defined using the deployment descriptor (Container-Managed Transaction Demarcation), but it is possible to define by hand the transaction scope in the code using the Java Transaction API (Bean-Managed Transaction Demarcation). In the next steps of this work we will tackle this aspect too.

Furthermore, we will also include other kinds of elements, such as JS Pages, into the Presentation Layer [7] and we will integrate them into the Server Layers visualization. We also plan to evaluate the relation between JEAs and databases linking methods to the tables that they access.

Another important step will be to use the information contained into the file version repository to trace the evolution of software elements. Thanks to this information it will be possible to create novel visualizations to more easily compare the same perspective through different versions.

EJB version 3.0 [19] differs significantly from version 2.1. It will be necessary to take into account all these differences in order to extend the same analysis to EJB 3.0.

This effort represents an initial step to create a set of techniques and instruments to expose problems and to investigate the elements composing JEAs. The final objective is to build a generic and extensible representation of JEAs to enable their analysis. The effort spent to analyze JEAs will be incorporated in this larger representation. Thanks to the descriptions ranging from course-grain to fine-grain it will be possible to analyze large and heterogeneous EAs without losing the possibility to perform analysis on specific aspects such as transaction scope.

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) and the financial support of CHOOSE, the Swiss Group for Object-Oriented Systems and Environments. We also gratefully acknowledge the contribution of our (anonymous) industrial partner for providing the case study used in this work. We thank David Röthlisberger, Jorge Ressia and Erwann Wernli for their comments on drafts of this paper.

References

[1] L. G. DeMichiel, "Enterprise JavaBeans specification, version 2.1," Sun Microsystems, Nov. 2003.

[2] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of ISPSE 2000)*. IEEE Computer Society Press, pp. 157–167.

[3] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of ESEC/FSE 2005*. New York NY: ACM Press, pp. 1–10, invited paper.

[4] E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock, "The J2EE 1.4 tutorial," Dec. 2005.

[5] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of ICSM 2005*, pp. 77–80, tool demo.

[6] 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.

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

[8] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[9] G. C. Murphy and D. Notkin, "Reengineering with reflexion models: A case study," *IEEE Computer*, vol. 8, pp. 29–36, 1997.

[10] C. Marinescu, "Identification of design roles for the assessment of design quality in enterprise applications," in *Proceedings of ICPC 2006*. Los Alamitos CA: IEEE Computer Society Press, pp. 169–180.

[11] C. Marinescu and I. Jurca, "A meta-model for enterprise applications," in *SYNASC '06: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 187–194.

[12] J.-F. Girard and R. Koschke, "Finding components in a hierarchy of modules: a step towards architectural understanding," in *ICSM*. IEEE Press, 1997.

[13] 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.

[14] R. Koschke and D. Simon, "Hierarchical reflexion models," in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society, 2003, p. 36.

[15] A. Christl, R. Koschke, and M.-A. Storey, "Equipping the reflexion method with automated clustering," in *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, 2005, pp. 89–98.

[16] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views — a case study," *Journal of Computer Languages, Systems and Structures*, vol. 32, no. 2, pp. 140–156, 2006.

[17] J. Fabry, "Transaction management in EJBs: Better separation of concerns with AOP," in *Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Y. Coady and D. Lorenz, Eds., Mar. 2004, pp. 20–25.

[18] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2002, pp. 406–416.

[19] M. K. Linda DeMichiel, "JSR 220: Enterprise JavaBeans specification, version 3.0," Sun Microsystems, May 2006.