Pattern-based Modifiability Analysis of EJB Architectures

Jorge Andrés Díaz Pace and Alejandro Zunino
ISISTAN Research Institute-CONICET, Argentina
Email: {adiaz,azunino}@exa.unicen.edu.ar

Abstract—Over the last years, several techniques to evaluate modifiability of software architectures have been developed. One of such techniques is change impact analysis (CIA), which aids developers in assessing the effects of change scenarios on architectural modules. However, CIA does not take into account the pattern structures behind those modules. In architectural frameworks, such as the Enterprise Java Beans (EJB) architecture, the use of patterns is a key practice to achieve modifiability goals. Although patterns can be easily understood individually, when an application combines several pattern instances the analysis is not straightforward. In practice, many EJB designs are assessed in an ad-hoc manner and relying on the developers’ experience. A way of dealing with this problem is through the integration of modifiability analysis models and patterns. We propose a knowledge-based approach that explicitly links the EJB patterns to a scenario-based analysis for multi-tier architectures. Specifically, we have developed a modifiability reasoning framework that reifies the EJB patterns present in a given design solution and, for a set of predetermined scenarios, the reasoning framework identifies which architectural elements can be affected by the scenarios. The reasoning framework outputs metrics for each of the scenarios regarding specific EJB tiers. The main contribution of this approach is that assists developers to evaluate EJB alternatives, providing quantitative information about the modifiability implications of their decisions. A preliminary evaluation has shown that the reasoning framework is viable to analyze EJB designs.

Index Terms—Component architectures, Design for quality, Software design, Software maintenance, Software reusability

I. INTRODUCTION

The quality-attribute evaluation of software architectures is recognized as an important activity to mitigate risks, evaluate design alternatives, and avoid premature (and costly) implementation commitments. Modifiability is a quality attribute that refers to the ability of a given system to accommodate changes. Several analysis and design mechanisms for modifiability exist [1], [5]. On the design side, architectural patterns and tactics (e.g., layers, intermediaries, etc.) can be used to build a solution that promotes certain types of changes [3]. On the analysis side, techniques such as change impact analysis (CIA) help to estimate the impact of a given change on the architecture [7], [5]. Conventional approaches for modifiability assessment [18], [5], [6] are based on scenarios and can also include a CIA model to quantify the costs of changing the architectural elements associated to the scenarios. However, such approaches seldom articulate the patterns used by the architecture with the CIA model. Pattern-based reviews of software architectures [15] have been recently proposed as a lightweight alternative to reason about quality attributes. Although one can easily analyze the (modifiability) effects of individual patterns on a particular (change) scenario [2], assessing the combined effects of a set of patterns is still a challenging activity that needs an important dosage of architectural experience. To alleviate this problem, we envision a quality-attribute assessment schema in which architectural patterns and quality-attribute analysis models complement each other. Our proposal extends the concept of reasoning frameworks [4], [12] with knowledge about architectural patterns.

The case above is common in architectural frameworks such as the Enterprise Java Beans (EJB) architecture. The EJB technology provides a catalog of design patterns and guidelines to developers [16], [20], [11], [9], so that they can build modular applications with different levels of modifiability on top of a multi-tier architecture. Thus, the EJB architecture can be seen as a source of pattern-based architectural knowledge. Although this knowledge is very helpful for analyzing design alternatives, many EJB applications are still assessed in an ad-hoc manner and relying on the developers’ experience. Examples of modifiability-related design decisions are the usage of the Session Facade and Business Delegate patterns. The first one hides business object complexity and centralizes workflow handling. The second one decouples presentation and service tiers, providing a proxy interface to a service. Both patterns, when applied individually, result on increased modifiability, but when combined the resulting design might require twice the number of modifications to accommodate changes. Without a careful analysis, the situation often leads to “over-design”, where more patterns than actually needed are applied in a solution. Along this line, we argue that the EJB patterns must be interpreted in the context of scenarios, and furthermore, that a CIA model should provide the glue between patterns and scenarios.

In this article, we present an approach that links the EJB design patterns of the EJB 2.1 specification\(^1\) to a scenario-based modifiability analysis for multi-tier architectures. Specifically, we have developed a modifiability reasoning framework that leverages on the EJB patterns present in a given design solution and, for a set of predetermined change scenarios, the reasoning framework identifies which architectural elements are

\(^1\)http://www.oracle.com/technetwork/java/javaee/ejb/
\(^2\)Our approach can be applied to other architectural frameworks based on patterns, such as EJB 3 or Spring.
likely to be affected by each scenario. Internal to the reasoning framework, there is an engine equipped with modifiability-centric architectural rules about the EJB architecture. This engine processes the pattern instances and outputs metrics per scenario for the EJB tiers. The main contribution of this approach is that assists developers to evaluate EJB designs, and boosts the pattern-based architectural reasoning for modifiability. We have developed a prototype tool called ModEsT4EJB (Modifiability Estimation Tool for EJB) that implements the reasoning framework approach. We have applied this tool to alternative designs for a well-known EJB application, and the results of the tool have been confirmed by EJB experts.

The rest of the article is organized as follows. Section II introduces some key concepts and presents a model for tailoring the general modifiability analysis with EJB-related knowledge. Section III describes how the EJB analysis model is packaged as a reasoning framework. Section IV presents a preliminary evaluation of the approach. Section V discusses related work. Finally, section VI gives the conclusions and future work.

II. ARCHITECTURE-LEVEL MODIFIABILITY ANALYSIS

It is well-known that the software architecture of a system affects the system functionality with respect to quality attributes (e.g., performance, modifiability, security) [3]. In this context, architecture-level modifiability analysis refers to estimating the efforts of making different types of modifications on the system, based on the information provided by architectural structures. This analysis is performed in early design stages (in general, the system has not been yet implemented) and it is usually based on scenarios [5].

A quality-attribute scenario describes events that stimulate the system and states how the system should respond to those events [3], [19]. In a way, scenarios work as non-functional tests for the architecture. In the case of modifiability, the events capture specific changes that the system must accommodate. For instance, a typical modifiability scenario for EJB applications is “the developer wants to add new business logic to her application, without affecting the data model”. It is important to specify the criteria to decide whether a scenario is satisfied by a given design. Examples of criteria are: the cost of accommodating a given change, or the number of components/interfaces likely affected, among others.

If we think of a software architecture as a structure that realizes a collection of functions, then the degree of modifiability of an architecture depends on how these functions are allocated to architectural components and on how these components interact with each other. For instance, a layered architectural structure will have different modifiability properties than a Model-View-Controller structure. A common technique for analyzing modifiability is change impact analysis (CIA) [7]. CIA essentially treats the architecture structure as a graph in which the nodes correspond to “units of change” (e.g., functions, components, interfaces) while the arcs represent dependencies between the nodes (e.g., functional dependencies, data flows). Given a modification affecting a particular node, the dependencies of that node act like “change propagators” to other nodes. This is known as rippling effect. Different CIA models are possible, based on the types of nodes, dependencies, and propagation rules for the graph.

The modifiability analysis for software architectures discussed so far is general, as it is agnostic about the semantics of the components or the nature of their dependencies. This semantic knowledge is assumed to be provided by the architect. Departing from a given modifiability scenario, she must perform the following tasks: i) interpret the scenario to discover implied functions, ii) find the components that realize those functions, so as to know the components affected by the change, iii) estimate the rippling effects for those components, based on the types of dependencies from the affected components to other components, and iv) compute an overall measure of the change impact for the input scenario. For example, in the EJB scenario above, the architect can identify the responsibilities associated to the business logic, then map their effects to both the business and data tiers, and finally estimate the extent of the change in terms of Java classes, beans or interfaces.

A. Leveraging on Domain Knowledge: The EJB Architecture

The EJB architecture codifies design knowledge in the form of patterns and design guidelines [20], [11]. These assets are intended to support the development of applications on top of the reference architecture. Moreover, architects can benefit from these assets when performing a quality-attribute analysis. Based on our experience with EJB projects, we argue that a modifiability analysis can involve different, and incremental, levels of knowledge. The proposed levels are listed in Table I.

The EJB architecture prescribes that applications should follow a design organization into three main tiers (see Figure 1). We are interested in the Server-side Business Logic tier, which basically encompasses an EJB Container where different types of beans (e.g., Session Beans, Message-driven Beans, or Entity Beans) reside. There are various usage dependencies among

<table>
<thead>
<tr>
<th>Level of Knowledge</th>
<th>Description</th>
<th>Architectural Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. General</td>
<td>Agnostic about EJB technologies</td>
<td>Architectural components and their dependencies</td>
</tr>
<tr>
<td>2. EJB-aware</td>
<td>The overall architectural structure and design prescriptions of the EJB framework</td>
<td>EJB tiers, types of Beans and their allowed dependencies</td>
</tr>
<tr>
<td>3. EJB patterns</td>
<td>Catalog of EJB patterns and their roles within each of the EJB tiers</td>
<td>EJB patterns, Java classes and methods</td>
</tr>
<tr>
<td>4. EJB pattern instances</td>
<td>Guidelines for instantiating and combining the EJB patterns</td>
<td>EJB patterns, Java classes and methods</td>
</tr>
</tbody>
</table>
the beans, as shown by the arrows. If an application is designed according to the EJB architectural prescriptions, we can refine the general CIA to an EJB-aware analysis. EJB-aware means that we can use the EJB tiers, types of beans and their dependencies in order to inform the working of CIA. For instance, if the interaction between a Web UI and the Session Beans needs to be changed, the initial impact set will comprise the Session Beans, while the candidate impact set might include the internal objects of the business tier, some of the Entity Beans, or even the data model. The rippling effect will depend on the actual dependencies between EJB elements. However, note from Figure 1 that the modifications should not affect the Message Driven Beans in our example, because the Web UI does not interact with that kind of beans.

The architectural reasoning can be richer by making CIA aware of EJB patterns (level 3 in Table I). In our example, it is common to use a Session Facade pattern to handle interactions with the EJB client and avoid exposing the details of the business model to the client. Let’s consider the well-known ShoppingCart application [16], in which a Session Facade is used for interactions with both the catalog and the cart components, as illustrated in Figure 2. Note that the sequence diagram cuts across the tiers of Figure 1, touching several modules within the Server-side Business Logic. For modifiability purposes, we have further divided this tier into sub-tiers, namely: Client, InteractionClient-EJBApplication, InternalObjects, and EntityBeans.

The Client sub-tier refers to the applications that are external to the EJB container, such as JSPs, Servlets, etc. InteractionClient-EJBApplication includes those Session Beans, Message Driven Beans and Internal Objects that are directly accessed by client applications. These beans provide processing services to the clients and act as entry points to the EJB application. InternalObjects are plain Java classes that provide functionality to the tier InteractionClient-EJBApplication, but remain hidden from the client. At last, the EntityBeans capture the data and accessors needed by the clients. These EntityBeans are persisted in a database. In general, the EJB patterns contribute information to refine the analysis of InteractionClient-EJBApplication and EntityBeans.

In this schema, the outcome of a modifiability analysis will be that the classes playing the Facade role (within the InteractionClient-EJBApplication sub-tier) will mainly absorb the changes from the Web UI. Since the Facade relies on additional classes to implement the service functionality (other Session, Message or Entity Beans), those classes will be candidates for change rippling effects. In our example in Figure 2, at least one of the Session Beans implementing the Facade pattern will be likely modified if a business logic change (without altering inputs/outputs) is introduced. On the other hand, as the Facade absorbs changes, InternalObjects as well as EntityBeans will not be subject to any changes.

There may be subtle modifiability issues, depending on how the Session Facade pattern is instantiated. A design guideline for applying Session Facades is that related use cases should use a single Facade instance, instead of having a Facade for each individual use case. This guideline tries to avoid clients having to manage too many EJB references. Along this line, we can extend our pattern-based analysis to include the specific elements that materialize each pattern in the application design. The latter analysis relies on pattern instantiation information, and corresponds to level 4 in Table I. Specifically, we take into account the number of application classes implementing a Facade and the methods implementing the Facade service(s). Continuing with the ShoppingCart example, a change in the Business Logic will likely affect at least one method in one of the Session Facades. If the change affects the Cart, and since it relies on the OrderProcessor Message Driven Bean, changes in the Cart may cause changes in the OrderProcessor. Note that a change in the Business Logic is much more likely to affect any of the Session Beans, than the Message Driven Bean. This will occur only if the parts of the Cart that rely on the OrderProcessor Bean are affected.

As a corollary to this section, CIA is very hard to perform manually, specially when EJBs are combined and several patterns are involved in a change.
Table II
Predefined Modifiability Scenarios for the Business Logic Tier

<table>
<thead>
<tr>
<th>ID</th>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Change the business logic</td>
<td>It refers to modifying the computations that implement a business use case (e.g., modifications in the steps that realize the use case), without changing the input/output data for that use case.</td>
</tr>
<tr>
<td>E2</td>
<td>Change exception handling policies in the EJB container</td>
<td>It involves changes in the actions taken in response to certain exceptions, when triggered by the EJB container. These exceptions are usually associated to a service invocation.</td>
</tr>
<tr>
<td>E3</td>
<td>Change exception handling policies at the application level</td>
<td>It involves changes in the types of exceptions (and the corresponding actions) that can be triggered by application services.</td>
</tr>
<tr>
<td>E4</td>
<td>Add new functionality</td>
<td>It captures new business logic being added to the application, without affecting the data model.</td>
</tr>
<tr>
<td>E5</td>
<td>Modify the internal objects of the EJB application</td>
<td>It covers modifications to components such as EJBs and other classes that do not interact directly with the client application.</td>
</tr>
</tbody>
</table>

B. A Scenario-based Perspective of EJB patterns

The use of EJB patterns helps to assess the modifiability of a given EJB application. Nonetheless, the role played by these patterns with respect to the propagation (or isolation) of changes is dependent on the types of changes being considered. From the previous discussion, we have that scenarios are common yardsticks for expressing those changes, and assessing thereof the suitability of an architecture. The EJB architecture is designed to support predefined types of changes, although they are not explicitly captured as scenarios. One goal of our work is to have concrete modifiability scenarios to enable analysts to track down changes on the EJB architecture (e.g., tiers, patterns, classes, etc.). After an examination of the Server-side Business Logic tier, we have distilled five representative scenarios, which are listed in Table II. Note that these scenarios are not meant to cover all possible changes of an EJB application. For instance, the case of adding new data structures is left out of our list, because we do not consider it as an architecturally-significant change.4.

Each of the scenarios provides the context to interpret certain change within the Server-side Business Logic tier. Moreover, we can associate every scenario with specific EJB patterns. This is where the knowledge levels of Table I become useful, particularly the last two levels. To this end, Figure 1 depicts the mapping of several EJB patterns to the sub-tiers of Server-side Business Logic. We see the EJB patterns used by an application as the vehicles that connect the modifiability scenarios of Table II with the CIA technique. That is, we have specialized CIA to produce scenario-based results in terms of EJB elements. In some cases, this analysis only says whether a sub-tier is likely to be affected by a scenario. In other cases, the analysis also estimates the number of beans/classes to be affected by the scenario. Note that our emphasis is not on precise results, but rather on “trends” that can support the assessment of EJB design alternatives.

For example, in the design of the ShoppingCart application using a Session Facade, the main component is a stateful session bean (class Cart) that represents the cart itself and keeps tracks of each customer’s purchases as shown in Figure 2. For the use case “add product to the cart”, the main service in the facade is the method purchase(). Additionally, let’s assume the Session Facade is used along with a Message Facade. The combined analysis for scenarios E1 to E5 is shown in Figure 5 (right panel). For scenario E1 (about business logic changes), the analysis indicates that neither the Client nor the Entity Beans should be modified, if there is no shared data between sub-tiers. Certainly, changes are expected in the InteractionClient-EJBApplication sub-tier, and more specifically, these changes will affect the session beans. On the other hand, the probability of a change affecting the Message Bean is very low. In the case of a different scenario, such as E3 (about changes in the exception handling policies at the application level), the modifiability analysis produces different results. There are likely changes in the Session Beans and Message Driven Beans (within the InteractionClient-EJBApplication sub-tier), but also in the Client and InternalObjects sub-tiers.

III. A Reasoning Framework Approach

The CIA model previously outlined integrates the architectural knowledge provided by EJB patterns with scenario-based analysis techniques. However, developers that are not modifiability experts might still find this model difficult to apply to her EJB designs, or might be overwhelmed by the information required to perform the analysis. For this reason, it is necessary to package the model in such a way its essential aspects are only exposed.

We envision a schema in which the developer is assisted by an automated analyzer, which takes both the EJB patterns present in the developer’s application and a set of target modifiability scenarios as inputs, but makes the CIA transparent to the developer. To this end, a useful abstraction is the notion of reasoning frameworks [4]. A reasoning framework encapsulates the analysis/design techniques needed to understand and estimate the behavior of a system with respect to a quality attribute, so that this knowledge can be (re-)used by non-experts. Reasoning frameworks have been developed for performance, reliability, or security, among others [21].

Specifically, a reasoning framework consists of six building blocks, namely: a problem description, an analytic theory, a set of constraints imposed by that theory, a model of the system
Our reasoning framework is implemented by a Java tool called ModEst4EJB (Modifiability Estimation Tool for EJB). The setup of ModEst4EJB comprises predefined modifiability scenarios (Table II), a catalog of EJB patterns, and knowledge that relates these scenarios and patterns to the EJB architecture. The EJB patterns shown in Figure 1 are the ones currently supported by ModEst4EJB, and they were taken from [20]. The tool is extensible to incorporate new scenarios and EJB patterns. From a user’s perspective, the main features of the tool include: mapping of elements of UML class diagrams (e.g., classes and methods) to elements of EJB patterns, selection of modifiability scenarios, and visualization of analysis results by means of tables. In the sequel, we explain how the interpretation and evaluation procedures of the reasoning framework are realized by our tool.

### A. Reification of EJB patterns

An EJB-based application often employs several EJB patterns. The interpretation procedure must reify those EJB patterns as first-class entities, in order to use them for the modifiability analysis. The reification characterizes each occurrence of an EJB pattern in terms of constitutive elements. We follow the approach proposed in [14], in which a pattern is seen as a “collection of roles that collaborate to achieve specific concerns”. A pattern structure is instantiated by binding all its mandatory roles to concrete design elements (e.g., classes, methods). For instance, the Session Facade specifies a role for the facade class, a role for the method that implements each service, and a client role for the class that invokes the facade. All these roles must be bound to specific classes/methods. ModEst4EJB provides a graphical wizard for the developer to select a given EJB pattern and bind its roles to Java classes/methods of her application. Figure 4 shows the reification of the Session Facade for the ShoppingCart application. In general, these classes/methods might come from the application code or from a UML class diagram.

The roles of a given pattern have logical relations among them that constraint the bindings of roles to classes/methods, so as to ensure a correct instantiation of the pattern as a whole. Two important constraints are type and cardinality. The type constraint enforces that a role is only played by elements of given type. In our example, the facade role must be played by a Java class. The cardinality constraint refers to maximum and minimum number of players allowed for a given role. In our example, the facade role must be bound exactly to one class, but the client role can be played by one or more methods, which might be beans or just simple classes. Roles can also have subordinate relations. A typical example is the relation between a role with a ‘class’ type and a role with a ‘method’ type. If a class role is bound, it is mandatory to bind all its method roles as well. ModEst4EJB checks that all the pattern roles are fulfilled, and that the role constraints are not violated. The same EJB pattern can be instantiated multiple times.

### B. Assessment of Modifiability Scenarios

As mentioned in section II-B, the different EJB patterns are mapped to specific sub-tiers of the architecture. Furthermore, the roles that describe the patterns are mapped to the corresponding sub-tiers. This information is stored by ModEst4EJB in the so-called modifiability tables. There is one modifiability table per scenario. Each table can be seen as a “horizontal slice” on Figure 1, which looks at the EJB patterns according to the sub-tiers in which they operate from the perspective of a particular scenario. For the sub-tiers InteractionClient-
EJBApplication and EntityBeans, the tables can discriminate instances of bean types, Java classes and methods.

The evaluation procedure generates modifiability metrics at two levels of granularity. The coarse-grained evaluation analyzes the likelihood of change of every sub-tier, based on the intent of the patterns (previously reified from the application) and the rippling effects to other patterns/components. The results of this evaluation are expressed as a confidence value that indicates whether a sub-tier will be always, probably, or never impacted by modifications. The fine-grained evaluation estimates the number of design elements (e.g., beans, classes, methods, etc.) affected by the change, again based on the intent of the patterns. The evaluations of sub-tiers and design elements are both performed per scenario. Figure 5 shows how the evaluation metrics looks like in ModEsT4EJB.

Let’s consider the assessment of scenario E1 about changes in the business logic (Table II). Let’s also consider the ShoppingCart application using the Session Facade and Message Facade patterns to deal with the InteractionClient-EJBApplication sub-tier. The estimation of the number of
session beans potentially affected by E1 takes all the beans that pertain to the two pattern instances. This result can be shown by the tool as the sum of the corresponding beans (i.e., the worst-case estimation), or averaged per service.

The rules for estimating the change likelihood of a sub-tier are as follows. If all the pattern instances for that sub-tier are affected by the scenario under analysis, then the value of the sub-tier is marked as 'sure' for that scenario. If none of the pattern instances on the same sub-tier are affected by the scenario, then the value of the sub-tier is marked as 'no'. Otherwise, the value of the sub-tier is marked as 'probably'.

ModEst4EJB allows for the customization of the evaluation rules for design elements and sub-tiers, in order to consider a deeper analysis of the semantics of the EJB patterns.

IV. EVALUATION

To judge the utility of our reasoning framework, we considered the well-known ShoppingCart application, which has functionality for buying items via Internet. Specifically, we exercised ModEst4EJB with two solutions, both implementing the same use cases but through alternative bean structures. The goal of the evaluation was to assess the modifiability implications of applying specific EJB patterns.

A. Two Design Alternatives for the Shopping Cart

The first design, used as a motivating example in the previous sections, is based on a published solution [22], which follows from the EJB specification. The main class diagram is depicted in Figure 7 (left side), while the corresponding sequence diagram is that of Figure 2. The persistence model is based on a set of entity beans that encapsulate the application data, do not contain business logic and are not accessible by clients outside the EJB container. The main beans are: Customer, Order, Line-Item and Product. The business logic and the data access are dealt with on the server side by means of a Session Facade. The main component realizing the business logic is the stateful session bean representing the cart. In addition, there is a message-driven bean responsible for interacting with the payment systems and for notifying the customers about the results of their transactions. This functionality was implemented with a Message Facade, so as to simplify the customers’ transactions. The second design, in turn, is a variant that uses an HttpSession object to keep information about a user navigating several Web pages and also to store data related to the user’s session. This mechanism is based on the Java Servlets technology. The main class diagram is depicted in Figure 7 (right side), and the corresponding sequence diagram is depicted in Figure 6. The main difference with the first design is that the stateful session bean representing the cart is replaced by a session object that resides on the EJB client (server-side presentation) and has direct access to the entity beans, as implemented in [24].

The functionality of the cart is implemented by the purchase() method of the Order entity bean, which works as an Entity Facade pattern. A local client executed on the same container as the EJBs is in charge of managing the products. The Catalog session bean is used to access the list of products, and this bean interacts directly with the entity beans. The EJB pattern applied here is Direct Interaction with Entity Beans. This design has the disadvantage of mixing concerns related to the business logic with persistence-related concerns. For the parts in common with the first design, the second design uses instances of the Session and Message Facade patterns.

The mapping of the first design solution in ModEst4EJB required 24 services: 23 of them were Session Facades, and the remaining was a Message Facade. This mapping involved 5 application classes: 4 session beans and 1 message-driven bean with their corresponding methods. The mapping of the second design solution required 16 services: 11 of them were Session Facades, 1 Message Facade, 1 Entity Facade, and 3 Direct Interactions with Entity Beans. This mapping involved 6 application classes: 3 session beans, 1 message-driven bean, and 2 entity beans, with their corresponding methods.

Table III shows the metrics computed by ModEst4EJB for the two design solutions (these metrics are the same of Figure 5). The comparison of results is discussed below.

- **Client** sub-tier: For this tier, the tool estimates likelihood of impact rather than detailed metrics. The consequences of the first design are mainly noticed in scenarios E2, E3 and E4, which talk about changes in exception handling or functional additions. However, the use of the Session and the Message Facade patterns isolates modifications on the business logic or the internal objects used by the beans. On the contrary, the second design is affected by scenario E1, because the client-side functionality is coupled to the server-side business logic. Specifically, the direct access to the beans is the root of the problem. Like the first design, the second design is insensitive to scenario E5, which is related to internal objects.

- **ClientInteraction-EJBApplication** sub-tier: Since many of the patterns supported by ModEst4EJB operate on this tier, it is possible to compute fine-grained metrics.
Note that the five scenarios are relevant to both design alternatives, but there are differences in the values of the metrics. Most of the metrics of the first design are higher than those for the second one, with the exception of the metrics related to Message Driven Bean classes to be modified that show even results. This trend is due to the considerable number of beans of the first design, which makes highly probable that no matter the change being considered, it will affect those beans (in the worst case, several beans will be modified). In the second design, the business logic is organized differently, and the number of beans is lower when compared to the second design. Still, some changes in the client-side of the application are not reflected in the analysis of the tool.

- **InternalObjects** sub-tier: Like in the first sub-tier, the results here are given in terms of likelihood of change. This sub-tier responds equally to all the scenarios for both design alternatives. Scenarios E1 and E2 remain unaffected, because they refer to changes either in services offered by the application or in the exception handling policies of the EJB container, which are both independent from the implementation of the internal objects. However, the changes specified by scenarios E3 and E4 might ripple to the internal objects, in case new invocations to those objects are required. Scenario E5 speaks directly to the internal objects, so they are surely modified.

- **EntityBeans** sub-tier: Similarly to the case of the second sub-tier, ModEsT4EJB outputs detailed metrics. The first design fulfills the modifiability issues specified by the five scenarios, mainly due to the application of the Session Facade and Message Facade patterns. Consequently, the entity beans are considered unaffected by the changes. The second design, in turn, shows a different situation, in which four of the scenarios impact the entity beans, with the exception of scenario E2 (which has to do with the EJB container). The situation is explained by the fact that the Order entity bean contains business logic, and the changes might propagate to other beans. Nonetheless, the probability of changes in the beans is low.

The analysis above clearly shows that not all design solutions (and their constitutive patterns) are created equal with respect to modifiability scenarios.

**B. Lessons Learned and Limitations**

Although the results produced by our reasoning framework are approximate, they work well as early predictors of the pros and cons of different design decisions regarding modifiability. The connection of these decisions with EJB patterns is a plus in our approach. In the ShoppingCart case-study, the metrics obtained for the two designs were consistent with the rationale for those kinds of solutions given in the literature. An interesting side-effect of the reasoning framework assessment is that of conformance checking. That is, developers get aware of discrepancies between the actual pattern-based design and what is prescribed by the EJB architecture.

In addition, we applied ModEsT4EJB to a real EJB-based application that supports the customer services division (e.g., user accounts, and credit card issuing) of a regional financial company. The company faced a typical quality-attribute tradeoff: how to support changes in the business logic without compromising performance when processing users’ requests. We performed a scenario-based analysis of candidate EJB designs for the business tier, to help the development team to elaborate a ranking of solutions. The developers were satisfied with the assessment, and the findings were corroborated by EJB experts. The developers also liked the fact that the scenario values were directly traceable to EJB patterns, since the team uses a pattern vocabulary in its daily work. We also observed, during a sub-system migration from EJB 2.1 to EJB 3, the validity of the core ideas of our approach.

A drawback of the ModEsT4EJB tool is the effort needed to identify the pattern instances in an application, before the evaluation can proceed. This aspect was particularly costly in the case-study above, in which a large set of services had to be manually mapped to EJB patterns. A limitation of the reasoning framework is the determination of the impact set, that is, all the elements potentially affected by a given scenario. The impact set used by the evaluation procedure only includes those elements that are bound to pattern roles and are within the boundaries of the business logic tier. Therefore, there might be other elements (e.g., helper classes, client-side classes, etc.) that have dependencies with the patterns but are not factored in when computing the metrics. A more complete analysis of design dependencies (e.g., based on probabilistic models) can alleviate this problem. Another issue related to dependency analysis is that our version of CIA currently reasons about rippling effects using simple rules. Our goal with these rules was to quickly determine if a sub-tier would be impacted by a scenario (i.e., the worst case), based on the number of modified pattern elements as hints. Certainly, the evaluation rules can be more complex, and take into account both the semantics of the patterns and their dependencies to other design elements.

The main threat to validity of our approach is that it does not guarantee that all the elements and dependencies are covered by the CIA. Another threat to validity is that the links between patterns and scenarios are established by EJB experts in a subjective manner. At last, the metrics computed by the reasoning framework might be biased, in part because of the two previous threats, but also because an empirical validation of the estimates against actual code changes is still pending.

**V. RELATED WORK**

Over the last years, several techniques for analyzing software architectures have been developed, such as ALMA, SAAM, ATAM, or PBAR [5], [18], [15], [19], among others. Most of these techniques are general in the sense that they can address different quality attributes. A few of them are specific to modifiability, as it is the case of ALMA.

7The details of this case-study are omitted in the article, due to space reasons and confidentiality agreements.
The SAAM [18] is a scenario-based technique for assessing individual quality attributes, which leverages on the architectural patterns used to build the architecture, and for probing their effects on those quality attributes. The ATAM [19] is an evolution of SAAM for multiple quality attributes that permits the identification of quality-attribute tradeoffs, risks and no-risk. ATAM is also scenario-based, and relies on studying the architectural patterns and tactics used to satisfy the key system scenarios. Both SAAM and ATAM were conceived as domain-independent methods, and do not incorporate knowledge like the one provided by the EJB framework. Perceived drawbacks of ATAM are the level of formality of its activities and the amount of information required for a review, which add up to the effort needed to carry out the evaluation. In contrast, PBAR [15] was conceived as a lightweight method for architectural reviews, whose distinctive feature is the identification of the architectural patterns (as they appear in the architecture) in order to support quality-attribute analyses. The reliance of patterns by PBAR is close to our approach, but PBAR does not include domain-specific knowledge to inform the evaluation. Another difference of our reasoning framework with existing evaluation techniques, with the exception perhaps of PBAR, is that they are oriented to work with high-level architectural views (e.g., modules, components and connectors) rather than with detailed design views involving classes and methods. Our approach departs from a high-level multi-tier architecture but also maps this architecture to classes via the EJB patterns.

Some researchers [2] have proposed design methods that articulate reasoning frameworks with quality-attribute scenarios, architectural patterns and tactics. There is an interesting study [2] about the relations of modifiability tactics with an existing catalog of architectural patterns. However, this study is not yet packaged as a reasoning framework that can be applied by non-experts. Approaches aligned with the ideas of reasoning frameworks have been developed in [8], [13], [17]. ALMA [5] is an evaluation method focused on modifiability issues, and furthermore, it can provide quantitative predictions (for example, via metrics) about the modifiability of a system. ALMA can be seen as a combination of SAAM with a modifiability reasoning framework. The method is scenario-based with support for different types of change scenarios, and the evaluation is a kind of change impact analysis. Unlike the methods above, and also our approach, ALMA does not consider information about patterns.

Stafford et al. [23] proposed an automated technique for architecture dependency analysis, that builds graphs of architectural components and captures their static and behavioral
relationships. The approach is implemented on top of an architectural description language, and it serves mainly to software maintenance analysis. Given a particular concern, a tool builds dependency graphs that support architects in the navigation and analysis of the set of components (also known as a slice) related to the concern. The notion of slice bears similarities with a modifiability scenario, but the slices do not take into account pattern-related structures though.

VI. CONCLUSIONS

We have presented a reasoning framework approach for analyzing modifiability scenarios in EJB applications. A novel aspect of this approach is the linkage of change impact analysis (CIA) techniques with architectural knowledge in the form of EJB patterns. As a result, EJB designs can be assessed using a vocabulary of patterns (as developers normally do). Moreover, the reasoning process gets improved because the patterns used in the application are interpreted from different perspectives (i.e., scenarios) and backed up with modifiability metrics. The analysis-related knowledge is compiled by EJB experts and encapsulated within the reasoning framework, so that users can readily apply it as a black box.

As a proof-of-concept, we have developed a tool that implements our reasoning framework. In brief, the modifiability assessment of an EJB application involves three steps. First, assuming that an object-oriented design exists, the developer must identify the EJB patterns present in that design. Second, the developer selects her scenarios of interest and runs our EJB-specific version of CIA, one scenario at the time. Third, she gets a table summarizing all the findings with respect to the scenarios. A first evaluation of our tool approach in two case-studies has reported encouraging results. Nonetheless, the estimates of the reasoning framework are still to be validated with empirical studies of code changes.

We envision several lines of future work derived from our approach. Regarding the modifiability reasoning framework itself, four aspects can be improved. First, we envision a wizard that automatically identifies the pattern instances as the developer is creating her application within an IDE. Second, new modifiability scenarios and EJB patterns covering the three architectural tiers should be incorporated. Third, we are planning to enhance the current CIA with more precise dependency analyses, and thereof, refine our set of evaluation rules. Fourth, it should be possible to build a similar reasoning framework targeted to performance analysis, such as [10].

From a broader perspective, we think that our integrated approach (scenario-based analysis plus patterns) can facilitate the assessment of other architectural frameworks in domains such as cloud computing or mobile applications. In this way, developers can capitalize on existing architectural knowledge while promoting quality-informed design decisions.

ACKNOWLEDGMENTS

The authors would like to thank R. Guasti and C. Ramos, who contributed to the development of the ModEST4EJB prototype and the case-study. The authors also thank the anonymous reviewers for their comments to improve this article. This work has been partially supported by ANPCyT (Argentina) through Project PICT Bicentenario 2010-2247.

REFERENCES