A Component-oriented Metamodel for the Modernization of Software Applications

Luciano Baresi and Matteo Miraz
Politecnico di Milano – Dipartimento di Elettronica e Informazione
via Golgi 42, 20133 Milano, Italy
baresi|miraz@elet.polimi.it

***Abstract*—The modernization of a software system is a complex and expensive task and requires a deep understanding of the existing system. The capability of re-factoring a complex application into some high-level views is mandatory to elicit its structure and start localize possible changes. The high number of different implementation technologies imposes a *model-based*, neutral approach to reconstruct the structure and hide unnecessary details. OMG supports this view and proposes KDM (Knowledge Discovery Metamodel) as means to describe software systems in detail, but unfortunately KDM supports a component-oriented decomposition of the system of interest only partially.**

To bypass this limitation, the paper proposes the COMO (Component-Oriented MOdernization) metamodel to extend KDM, by borrowing recurring concepts from component-based solutions and software architectures, and to support a proper componentization of the system we want to modernize. The paper presents the main elements of the COMO metamodel and exemplifies them on a simple case study.

I. Introduction

Many software systems are becoming aged, and their maintenance costs are becoming higher and higher. Generally speaking, this is because the older a system becomes, the more expensive (and painful) changes are. Moreover developers are often forced to create ad-hoc patches that, even if seem to satisfy the new requirements, ball up the entire structure of the application, and violate the original design decisions. On the other end, software applications are important assets for any modern company that is usually very reluctant to get rid of its existing systems and introduce completely new ones. The trend is to keep software systems alive as long as possible, and also try to reuse them (their components) in a variety of different business situations.

Componentisation is key to reuse and it is also a means to ease maintenance, but components —and their interactions— are too often tangled within the system. The lack of a clear software architecture [1] hampers the actual reuse of the different parts, which are seldom shaped as fully independent entities. Unclear interactions complicate the redesign, and replacement, of a single part of the system since one needs to understand its functionality, the components it depends on, and also those that depend on it. This is why, even if there is clearly a point above which the cost of maintaining a component becomes higher than its value [2], we often tend to postpone its substitution. The clear and full understanding of what we want to change, usually not supported by available documentation, is often a barrier against more radical changes.

Traditional maintenance activities (corrective, perfective, and adaptive) are usually devoted to correct faults, to improve systems' performance, or to adapt to changed contexts. However, when the scope of these changes becomes wide, a thorough evolution of the system —often called *modernization*— becomes mandatory. Even after years of experience in programming languages, abstractions, and software processes, modernization is often carried out by reasoning at the level of the source code, maybe because of the scarce quality of available documentation, which is usually not aligned with implemented components. There is no real attempt to conceive a global —and sufficiently abstract— view of the system before starting to change it.

OMG wanted to fill this gap by proposing the ADM (Architecture Driven Modernization [3]) initiative, as means to address all the main aspects of the modernization of complex software systems. ADM proposes KDM (Knowledge Discovery Metamodel) as reference metamodel to produce significant models from any software artifact and get rid of implementation details. Unfortunately, KDM works at a quite low level, and does not provide constructs to conveniently render the architecture —components and connections— of a complex software system. In contrast, we think that modernization requires that components and interfaces, which are the elements of interest, be properly identified.

Our assumption is that modernization must start by reconciling, and rendering, the system's structure at architectural level. To this end, the paper proposes the COMO (Component-Oriented MOdernization) metamodel to extend KDM with the concepts of component and interface. By lifting the abstraction level (w.r.t. KDM), we can produce an adequate view of the entire system, and leverage the boundaries between components to better focus the subsequent modernization effort. At the same time, the compatibility with KDM ensures that each component be associated with the concrete set of programming elements that constitute its implementation and changes be propagated to the corresponding source code.

The COMO metamodel was used within the EC project MOMOCS[1] to render complex systems and foster their modernization. This paper reconstructs and presents the architecture of an open-source billing system widely used in the telco domain and also exploited in one of the project's demonstrators. The first results witness that the metamodel fully supports the elicitation of the software architecture, facilitates the understanding of the system, and thus ease its modernization. This is in line with the outcome of the EC project: the COMO metamodel allowed our partners to create a modernization suite able to help and support the modernization of complex systems.

The rest of the paper is organized as follows. Section II briefly presents KDM, which is the basis of our work. Section III introduces the case study and elicits the requirements behind the modernization of a complex system. Section IV introduces the COMO metamodel and Section V presents a preliminary evaluation of the proposal. Section VI compares it against the state of the art. Section VII concludes the paper.

II. Knowledge Discovery Metamodel

The Knowledge Discovery Metamodel (KDM, [4], [5]) defines a metamodel for the representation of existing software applications, the relationships between its parts, and those with the environment. KDM supports both object-oriented and procedural artifacts and provides them with a homogeneous low-level representation. It also fosters modernization through model-based transformations, and ensures that all changes be easily propagated to the source code, whose relationships with the model are kept through traceability links.

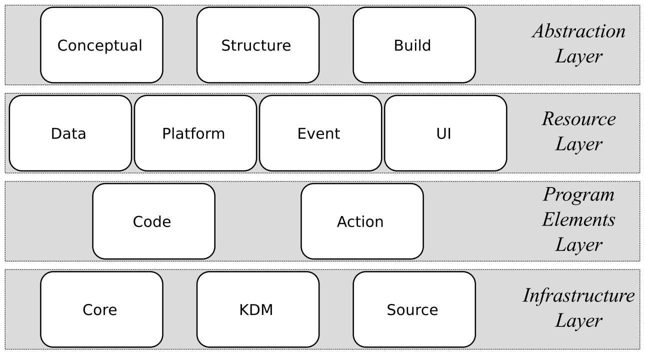

Figure 1. KDM organization.

The KDM metamodel is organized around the four layers of Figure 1, where each layer describes the application at a higher abstraction level. The first layer, called *Infrastructure Layer*, provides the root (abstract) elements, along with their common properties, for all KDM concepts: a `KDMModel` comprises `KDMElements` and `KDMRelationships`, which make any KDM model resemble an entity-relationship diagram. Moreover, each `KDMElement` can be linked to its corresponding physical artifacts (e.g., source files, images, or configuration options) to set the relationships between model elements and source code regions, and be able to propagate model-based transformations onto the relevant physical elements.

The *Program Elements Layer* specializes the elements above to describe the actual implementation of the system and provides means to render the program elements and their behavior. It abstracts from specific programming languages, and represents the low-level details of the system in terms of data types, callable units, and shared variables; their behavior is rendered by means of a sort of abstract syntax tree.

The last two layers enrich the basic model. The *Resource Layer* deals with the run-time resources used by a system. It supplies the elements to represent the run-time environment, which sometimes hinders the comprehension of the overall behavior of the system. Modeling the environment that hosts the system eases the understanding of subtle interactions between its parts: for example, a neat model of the locking system would allow us to detect deadlocks. This layer is also in charge of modeling the user interface, including possible compositions and allowed sequences of operations, and the organization of data. This way, we can accurately model complex data repositories, like record files, relational databases, or XML schema, along with the possible operations on them. The behavior of (parts of) the system is rendered by using state machines, which properly model the abstract states associated with the different resources.

The last layer (*Abstractions Layer*) provides the elements to represent domain- and application-specific abstractions. We can address the building phase of the system by accurately describing how to compile the source files and the artifacts the process generates. It is also possible to render the structure of the system by grouping basic KDM elements into layers, subsystems, and components[2]. The last abstraction supported by KDM borrows its key elements from *SBVR* (Semantics of Business Vocabulary and Business Rules) to represent the conceptual model of a system as *terms*, *facts*, and *rules*.

If we think of how (modernization) tools can exploit KDM, each tool can claim to be *L0* KDM-compliant. This means it must be able to understand the first two layers, that is, it must be able to cope with the overall structure, on one side, and the low-level programming constructs, along with their behavior, on the other side. The capability of understanding part of the other elements leads to *L1*

[1]www.viewzone.org/momocs/.

[2]A KDM component is more a container for lower level entities than an element of the software architecture of the system.

compatibility, while if the tool is able to deal with all the higher-level constructs, we have *L2* compatibility. All tools must preserve the parts they do not understand.

III. JBILLING

In this paper we use *JBilling*, which is an open-source billing system[3], as example of software application we want to modernize. This is an interesting example since its nature allows us to consider it a good representative of “average” (in terms of both quality and dimension) software projects and its size —even if not gigantic— needs attention. *JBilling* comprises 581 classes (91 KLOC) and 227 JSPs (14 KLOC). Moreover, it leverages the services provided by the J2EE application server that runs it, and the comprehension of the system is further hampered since the interactions among classes are mediated by it.

KDM allows us to create an integrated model of the whole system to precisely describe each class and the mediation provided by the application server. Figure 2 shows a UML-like simplified representation of the KDM model of *JBilling*. The high-level view comprises three parts: **Database**, which stores both users’ bills and the rules for calculating rates, **Server**, which hosts the billing logic and is in charge of the actual computations, and **Client**, which offers a Web-based administration console. Moreover, these three parts are implemented using different technologies: the Web-based subsystem uses STRUTS, the database storage is a layer of entity beans, and the logic is realized with session beans.

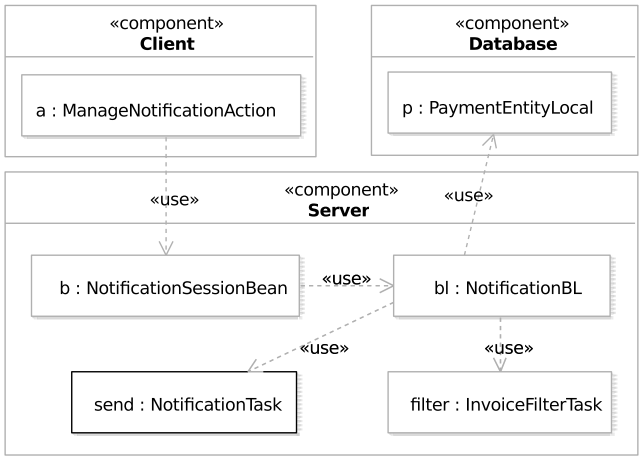

Figure 2. Partial view of JBilling’s architecture

Even if there are different features provided by JBilling, they all are organized around these three layers and follow a similar pattern. Each feature can be seen as a vertical subsystem (with respect to the three parts): for example, the figure concentrates on the internals of the notification subsystem. The user manages it through the Web-based administration client by using the `ManageNotificationAction` class, which in turn uses `NotificationSessionBean`. This is a facade to the notification business logic (`NotificationBL`) in charge of the real business decisions. This class uses `NotificationTask` and `InvoiceFilterTask` to perform some special-purpose operations, and `PaymentEntityLocal` to access the database.

This is all we can get from KDM: its high-level components are more boxes than real architectural components. As soon as the number of relationships between the elements contained in the different boxes increases, the readability and usefulness of such a model are quickly compromised. Even if technically feasible, it is unrealistic to start reasoning on the application by analyzing such a big, flat, and fine-graded model. KDM does not provide well-defined boundaries among its components (containers), which do not declare any interface, be it offered or required.

KDM provides a hierarchical representation of the system, but low-level `use` relationships can cross the boundaries of the different components. This leads to tangled dependencies, which hinder the comprehension of the system, and also precludes the reuse of the different parts. In contrast, a more component-oriented view would facilitate the analysis of the system (at architectural level), and help better scope changes. If we were able to reason in terms of cohesive elements, with precise offered and required interfaces, we could easily identify changes, predict their impact, and reason on substitutability and dependency mismatches. At the same time, components and interfaces must be linked to the concrete programming elements that form them. These links ensure that all architectural changes be directly reflected onto the source code.

IV. COMO METAMODEL

The COMO metamodel aims to overcome the limitations described above by combining KDM, that is, the state of the art metamodel for program modernization, with traditional concepts borrowed from software architectures. The former is good at creating the structural view of existing systems, while the latter are good at modeling the overall functional organization. Our metamodel combines the benefits of the two domains: it enriches KDM with the modeling elements required to fully render the software architecture of the system, and exploits it to correlate model-based changes to those on the source code.

As already said, KDM supports low-level programming constructs, domain-specific concepts, and high-level views. As for the latter, it offers the elements of Figure 3. `StructureModel` allows for the creation of different “structures” of the system. `KDMEntities` can be organized into a set of `StructureElements` by using the `groups` relation. Each `StructureElement` represents a set of

[3]Available at http://www.jbilling.com.

KDMEntities that has a precise role in the overall organization of the system. KDM also supports the hierarchical partitioning of the system by means of a ownership relation between different structural elements.

KDM provides several kinds of structure elements (e.g., subsystems, components, layers, etc.) to model complex structures by mixing them adequately. However, these elements are nothing but named containers, able to host and group together low-level elements. The meaning of these structure elements is misleading since they provide a hierarchical organization, but do not support any high-level abstraction since the relationships among them are defined in terms of low-level dependencies among low-level elements.

These structural views also come with severe limitations that hinder their actual usability. Since the structural division must be non-overlapping and strictly hierarchical, the reuse of a component is forbidden, but this is clearly against the idea of reuse, which suggests to use and share the same elements as many times as possible. Moreover, KDM does not explicitly define how different structure elements interact, and thus we cannot clearly detect what a component offers or what it requires from the others. For this reason, when we want to modify something, KDM forces us to perform a complete check of the system model, losing the benefits of a more disciplined and constrained approach.

To avoid all these limitations, the COMO metamodel enhances KDM with well-known forward engineering concepts to better accommodate the functional organization (architecture) of a system.

Figure 4 presents all the elements we introduce and specifies their KDM types (super classes). A ComponentModel hosts a COMO model and renders the architectural view of the system. For this purpose, it has a reference to all its ComponentElements, which can be either Components or Services. The former renders components, while the latter expresses the idea of functionality that can be offered or required by a component. There are also new types of relationships to enable a component to expose or require some services (through entities Offers and Requires, respectively), to interconnect components (element Connection), or to render delegation in composite components (relationships Exposed and Required).

Figure 5 concentrates on COMO components. A Component is a group of other KDM elements, thus it is able to host standard KDM entities, such as code elements and run-time resources. Technically, it is also able to host other components (since Component specializes KDMEntity in Figure 4), but to create composite components it is preferable to use our composition mechanism, detailed below. A component may be flagged as *external*, to mean it is provided by an external party, and it is not possible to modify it. For example, external components are bought from other companies or used remotely, like Web Services.

A Service renders a feature offered or required by a Component (using element Offers or Requires, respectively). When a Service is offered, we must provide both the DataTypes and ComputationalObjects declared by the Service. In a similar way, when a Component requires a Service, the Component depends on some of the Service's DataTypes and ComputationalObjects. A Service can contain particular data types, modeled in KDM by means of proper DataType elements, callable methods (ControlElement), and shared variables (DataElement). This way we can distinguish what a component offers publicly, with respect to its internals, and impose that when it is changed, we trigger an adequate check on the whole system.

Figure 6 shows how components can be assembled from other components. This composition mechanism should allow one to reuse the same component while forming different composite components, as well as to use multiple instances of the same component while forming the composite one. For example, let us consider the creation of a graphical user interface with predefined components. The component that renders a text field could be used in many different situations: it could be used several times within a single form, and it could also be used in other forms.

The ComponentPart element enables both kinds of reuse. Each part represents the use of a component, referenced by its type, inside another component. This structure fosters the reuse of components that can be hosted by different components as needed. Obviously, we must avoid cycles in the composition to avoid strange situations (e.g., component A is composed of B that in turn is composed of A), but we can create multiple parts with the same type inside the same composite component. Note that a composite component is a component, and thus it can also hosts arbitrary KDM entities. Mix (internal) components with additional code is important to provide more advanced services than those provided separately by the parts. This way one can model the code that it is necessary to glue the internal components, able for example to coordinate them and to adapt the different data formats. Real-world systems often use such a mechanism, and the metamodel renders it.

Once composite components are created, we can handle the services exposed or required by its internal components. In particular, it is mandatory to satisfy all services required by internal components, and it is also possible to export the services they provide. These relationships can be materialized in three different ways:

- An internal component requires a service not provided by another internal component. The request is propagated outside the composite component by using a Required relationship that connects the internal request with the external one. Since we can have several internal components of the same type within the same

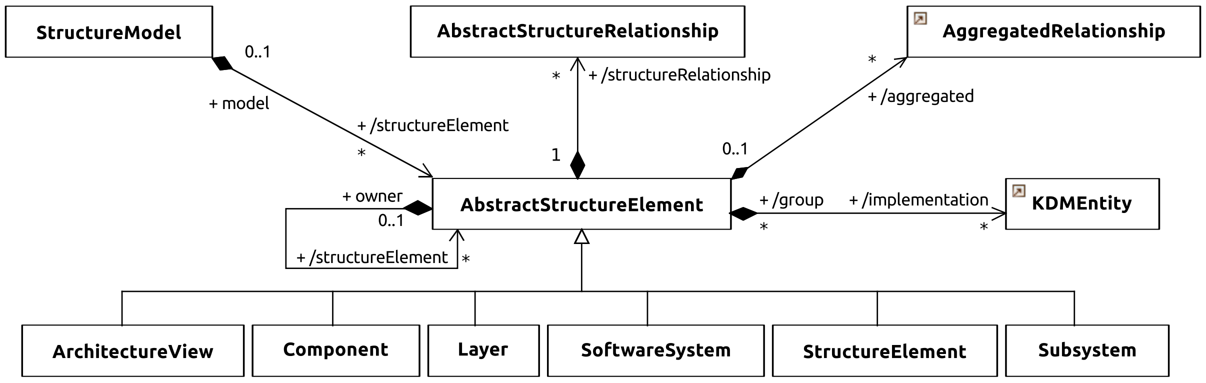

Figure 3. KDM structural elements.

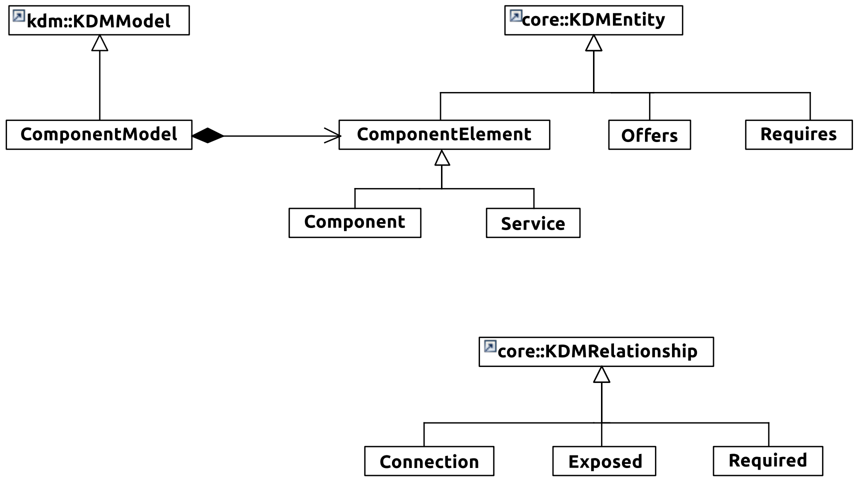

Figure 4. COMO: Main elements

composite component, we uniquely select the one of interest by adequately setting the `internalParts` attribute.

- A service offered by the composite component is provided by an internal component. To render this situation, we use the `Exposed` relationship by connecting the external offer to the actual internal provision. Again, `internalParts` uniquely identifies the internal component that provides the service.
- An internal component requires a service that is provided by another internal component. In this case, we need to connect the `Offers` and `Requires` entities involved by means of a `Connection`. This way, we uniquely identify how the interaction happens.

The elements presented so far model the static structure of the system, define boundaries among components, and specify that all possible interactions must pass through the predefined services (i.e., their interfaces). Now, we have a structure that better fits the needs of modernization efforts. The well-defined boundaries allow us to easily add new components, change existing ones (by paying attention to offered and required services), and replace those parts of the system that do not satisfy our requirements anymore. Moreover, since everything is tightly integrated into KDM, it is guaranteed that these modifications are directly mapped onto the real system (low-level entities)

The COMO metamodel also supports standard and widely used constraint languages, like OCL [6], to express constraints imposed by the architectural style. For example, we can easily define a layered architecture by specifying that components at layer n can only use services provided by layer $n-1$. Similarly, we can define more advanced constraints on the system like those, for example, that govern the interactions behind the Model-View-Controller pattern. However, when one is dealing with a complex system, s/he might want to relax some constraints so to enable further transformations. For this reason, we propose to create two categories of constraints: *strong* and *weak*. Strong constraints must always hold at each modernization step (i.e., after any transformation). Instead, weak constraints can be broken

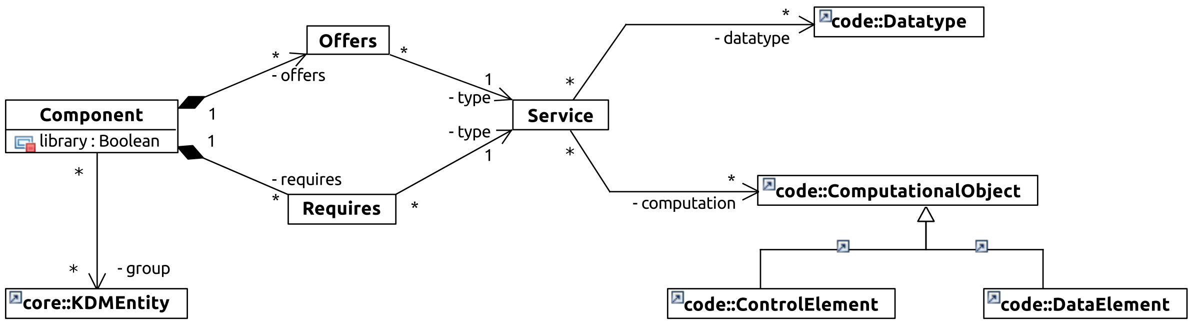

Figure 5. COMO: Components

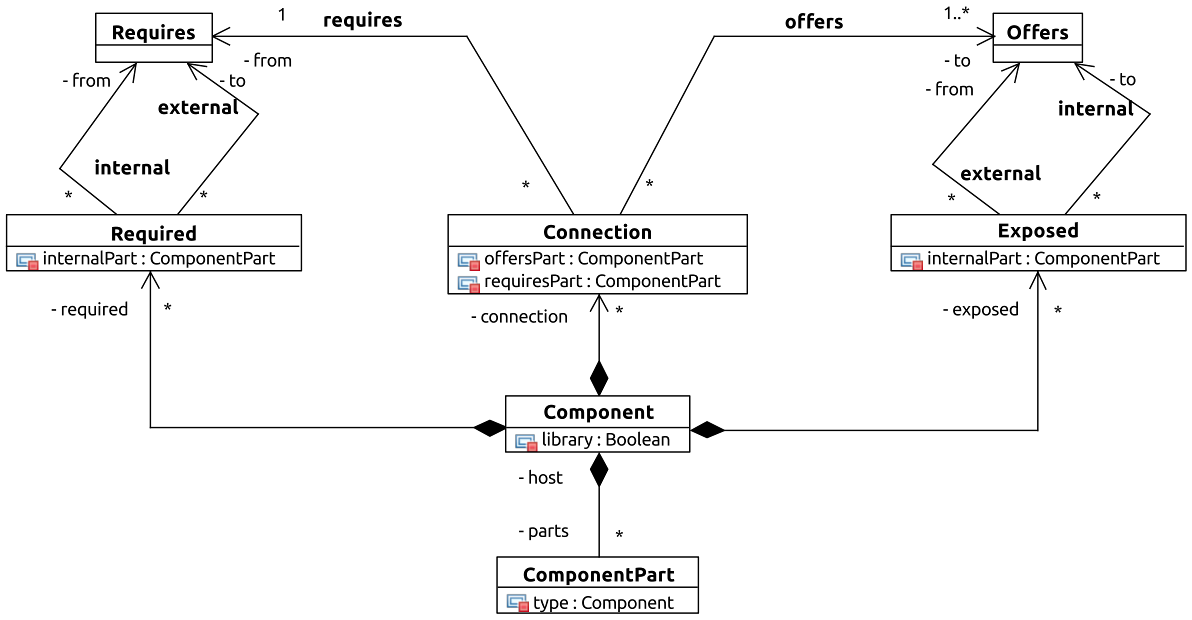

Figure 6. COMO: Composite Components

temporarily to enable further transformations of the system.

Albeit rendering the behavior of the system goes beyond the goal of this article, we propose to leverage KDM and use state machines. This way, one can model the reactions of components to method/operation invocations by specifying the effects on both their internal states and the other components (contacted through the required services). Additionally, one can use advanced reasoning techniques and simulations to better support component substitutability. If the whole system is modeled through a set of cooperating automata, it is possible to simulate its behavior and easily identify possible problems, even before performing the actual changes. Moreover, when one wants an alternative to a given component, s/he can reason on both syntactic properties (i.e., the set of offered/required interfaces), and desired behavior to foresee potential integration issues.

V. Preliminary evaluation

To demonstrate the applicability and usefulness of our approach, we implemented the COMO metamodel using EMF (Eclipse Modeling Framework [7]), and created a simple tool, on top of it, able to infer the COMO-like architecture of J2EE applications. The tool heavily leverages the strongly-typed J2EE architecture to understand the boundaries of each element and propose a first draft architecture. Unfortunately, and to the best of our knowledge, no tool is able to analyze the source files of an application and produce the equivalent KDM model. Since we were mainly interested in assessing our extensions, and due to this lack, we decided to derive the high-level elements provided by the COMO metamodel from the source code directly, but the same approach (algorithm) can be adopted by using information gathered from KDM. The result would be easier and neater, but out of the scope of our experiments.

We used the tool to assess the validity of our proposal by automatically creating the structural model of JBilling. Even if the prototype is raw, the results are interesting. The overall model of *JBilling* is represented, using a UML-like formalism, in Figure 7. The outermost component represents the overall JBilling application, as a customer can download from the web site. As expected, it provides service *Bill*. It has some unsolved dependencies, which means that it cannot be used as-is, and other components —able to solve those dependencies— are required. In particular, *JBilling* requires service *JPA*, able to manage the persistence of Java objects, service *Session*, which manages the life-cycle of session beans, and an interface *SQL* for accessing a relational database.

Digging down into the internal structure, JBilling is composed of three parts: `Client`, `Server`, and `Database`. The diagram explains the interactions among these components, since they are explicitly declared using services *DAO*, for data access, and *Management*, for administering the Server. The `Server` requires a service to manage session beans, provided by someone outside JBilling, and a service for data access, provided by the `Database` part. If such services are available, the application successfully provides both service *Bill* and service *Management*. Notice that the metamodel groups low-level dependencies among the internal elements of each component into high-level cohesive requirements, and thus simplifies the overall model.

We can also better define the internal structure of each part, and identify the sub-components able to manage the notifications. Each internal component contributes to the overall services that the containing component offers or requires; this is rendered in the diagram through dotted lines. For example, if we consider the `Server` component, part of its `Management` interface is provided by the `Notification Management` component.

The functionality provided by a J2EE application server is modeled by creating the *Persist* service, able to manage entity beans, the *Message* service, able to provide JMS functionality, the *Session* service, able to manage the life-cycle of session beans, and the *JSP* service, able to render dynamic web pages. We can also model an application server compliant with these specifications by creating a component that offers these services. In a similar way, we created models for the database, which offers an *SQL* interface, and Apache STRUTS, which provides the homonym interface leveraging on the *JSP* service. The application relies on some functionality provided by external components, thus the `JBillingEar` requires the corresponding services to calculate telephone bills, rendered through the *Bill* interface.

Using this model, one can effectively tackle the modernization problem at a higher level, narrowing the changes into well-defined boundaries and reasoning in terms of which component should be modified to satisfy the final requirements. The aim is to foster a component-oriented modernization approach. Initially users exploit the COMO metamodel, along with its KDM-compliant supporting tools, to define a first draft architecture of the application they want to modernize. Supporting tools let users refine automatically produced models by hand. All the results are stored in a knowledge repository, which provides versioning facilities, enhanced model search using ontologies, and a transformation library to support the actual modernization of the system.

After creating the initial model, ATL [8] transformations help modernize the application. These transformations help reassemble the structure of the system by merging, splitting, or deleting components. They can both work on the low-level KDM elements and on our components and interfaces. For example, it is possible to adapt the JBilling's `database` layer to the data abstraction currently used in the telephone company.

Users are also free to modify models by hand: the metamodel, and its constraints, ensure the consistency of produced results. Moreover, transformations can check whether these manual modifications on some components affect other neighbor elements. For example, we can easily query the repository for a component able to provide the required `session` interface, and find a proper application server. Since the retrieved component also provides the `JPA` interface, the rule checks the consistency among the parts and easily establishes the connection.

VI. Related Work

The proposals related to our COMO metamodel can be grouped into two main groups: model-based techniques and architectural-oriented solutions.

As for the first class, Strein et. al. [9] propose an extensible metamodel for program analysis, able also to serve as basis for a flexible modernization framework. They figure out that, when we are required to analyze a system, there is a set of principal tasks that must be accomplished: we need to extract information from the system, analyze it, and either show the results or perform a refactoring. This is true regardless the precise kind of analysis to perform; it only varies the level of abstraction of extracted information. Their proposal leverages the loose coupling among front-ends for programming languages, analysis techniques, and refactoring tools. Each part can be plugged independently of the others, allowing good reuse, and also ensuring that the system is able to adapt itself to unforeseen requirements. For these reasons, they propose a metamodel, inspired by *abstract syntax trees*, composed of three parts: a *front-end*

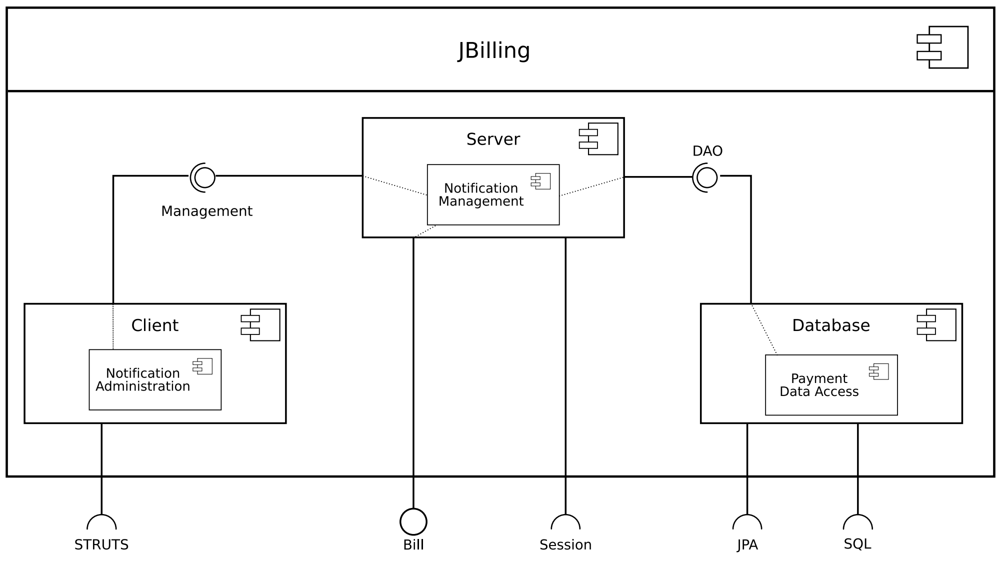

Figure 7. Component-based view of the case study

specific model produced by ad-hoc information extractors, a language-independent *common model* achieved by mapping front-end specific constructs by means of common equivalences, and some analysis-dependent *views* generated from the common model.

Obviously, we can have several concrete front-end models, one for each input language we understand, related to a common model of the system, on which each analysis can calculate its own view. The authors show the feasibility of a model-driven approach to foster the modernization of existing systems based on a language-transparent metamodel. They also demonstrate the efficiency and scalability of the proposal by allowing the resulting framework to incrementally update models. Nevertheless, this work does not help developers outline the structure of the system: their focus is to provide a generic metamodel, usable as basis for creating new analysis tool on systems. Moreover, the results of such analyses will not be integrated into the base system, and thus when the system is modified, (part of) the results of the analysis must be regenerated. For these reasons, it does not fit well with architectural models, which require a more careful management. Our approach integrates the architectural model with the model of the system and allows a better evolution of both.

Similarly, software architecture tries to lift the abstraction level on software systems, and shifts the focus from lines of code to coarse-grained architectural elements. Even if there is no universally accepted definition of what a software architecture is, Garlan and Shaw [10] say that:

> Software architecture involves the description of elements from which the systems are build, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Architectural description languages (ADLs) provide conceptual means to let designers focus on the conceptual architecture of their system without implementation details. We can use ADLs in different contexts, from the communication and understanding of a system, to advanced analysis techniques able to identify, for example, possible bottlenecks of the system. These different requirements led to the creation of many ADLs, surveyed in [11]. Garlan et al. tried to merge all main concepts in ACME [12], which is able to map architectural specifications from one ADL to another. The OMG introduced architectural concepts in UML2 [13], and now they are widely spread and used in many software design activities. These concepts also motivated *component-based development* [14] that aims to build systems by composing pre-existent components. Reasoning in terms of offered and required interfaces allows one to glue together existing components and (easily) release new applications.

Typically these architectural views on the system are successfully used in forward engineering. For example, Baresi et al. [15] present an approach to verify whether an architectural style is adequate to satisfy the elicited requirements, and if it can guarantee an adequate level of flexibility. Moreover, *ArchJava* [16] permits one to embody architectural models of the system within the actual implementation, and use high-level operations to modify the current structure of the system. Finally, Roshandel et al. [17] present a way to manage the architectural evolution of the system by combining a configuration management system and typical architectural concepts.

The research in this domain is also trying to infer the architecture from existing systems. One of the most successful approach is DiscoTect [1], which is able to identify the architectural structure of a system by knowing its architectural style and by monitoring its run-time behavior. The main problem of these approaches is their limited ability to handle the traditional backward-engineering issues.

VII. Conclusions and Future Work

This paper proposes the COMO metamodel as enhancement of the OMG's *Knowledge Discovery Metamodel* to fully support component-oriented modernization. We start by eliciting the main requirements behind the modernization of (complex) software applications, strengthening our idea to tackle this problem at component level, and then we introduce our metamodel.

Albeit the overall KDM structure is promising, we found that it can be improved with state of the art concepts borrowed from component-oriented development and software architecture. For this reason, the COMO metamodel is heavily based on components and interfaces and suggests how to represent all important information pieces. We suggest the declaration of reusable components that cooperate via well-defined interfaces. Preliminary results were promising since the metamodel allows us to model properly the key structural elements of the case study. Moreover the identification of precise interfaces, which mediate the interactions among components, allows us to adopt a higher point of view on the system and reason in terms of component substitutability.

For the future, we plan to enrich the COMO metamodel and provide the user with the ability to better render the architecture of the system, for example by considering UML ports. We also plan to refine the behavioral model of the system, to support advanced reasoning, and foresee potential integration issues. Finally, we would like to further elaborate general analysis techniques, able to semi-automatically identify components and interfaces, and better support the whole modernization process.

References

[1] B. R. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 454–466, 2006.

[2] Y. Tan and V. Mookerjee, "Comparing uniform and flexible policies for software maintenance and replacement," *IEEE Transactions on Software Engineering*, 2005.

[3] Object Management Group, "Architecture Driven Modernization (ADM)," http://adm.omg.org/, January 2006.

[4] ——, *Knowledge Discovery Meta-Model 1.2*, Needham (MA), USA, Jun. 2010, OMG doc. formal/2010-06-03.

[5] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[6] Object Management Group, *Object Constraint Language (OCL) Specification. Version 2.2*, Needham (MA), USA, Feb. 2010, OMG doc. formal/2010-02-01.

[7] "Eclipse Modeling Framework (EMF)," http://www.eclipse.org/modeling/emf/.

[8] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[9] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, "An extensible meta-model for program analysis," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 592–607, 2007.

[10] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[11] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[12] D. Garlan, R. T. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of CASCON'97*, Toronto, Ontario, November 1997, pp. 169–183.

[13] Object Management Group, *Unified Modeling Language 2.3 Superstructure Specification*, Needham (MA), USA, May 2010, OMG doc. formal/2010-05-05.

[14] K.-K. Lau and Z. Wang, "Software component models," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, 2007.

[15] L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Modeling and validation of service-oriented architectures: application vs. style," in *ESEC / SIGSOFT FSE*, 2003, pp. 68–77.

[16] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *ICSE*. ACM, 2002, pp. 187–197.

[17] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic, "Mae - a system model and environment for managing architectural evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 240–276, 2004.