Combining Model-Driven Development and Architectural Design in the AR3L Framework

David Ameller, Xavier Franch

Universitat Politècnica de Catalunya
{dameller, franch}@lsi.upc.edu

Abstract. Model-Driven Development (MDD) has the ultimate goal of producing code from some kind of analysis model, aiming at maximizing the automation of software development. However, it is difficult to inject in the MDD transformation process all the knowledge, skills and experiences that software architects have and which may be crucial when considering the satisfaction of system properties and constraints over the system-to-be. In this paper we propose the AR3L framework that integrates the software architect role into the MDD transformation process by recognizing the need of human interaction in an intermediate design phase that produces an architecture model. We focus on a particular domain, namely the development of Information Systems which, considering the current state of practice, determines the type of analysis model (object-oriented specification) and the target architecture pattern (3-layer architecture). We use responsibility-driven design as methodology driver. We present a proof-of-concept prototype implemented over the AndroMDA tool that shows the feasibility of our approach.

1 Introduction

Model-Driven Development (MDD) is a software development paradigm that has gained acceptance in the last years. According to [1], "Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing". Usually, the "real thing" is understood as the code, therefore most MDD approaches aim at producing some executable code as final result of a transformation process from some starting model of the system.

This development paradigm partly collides with the vision that software architects have about software system design. Apart from the functionalities described in the analysis model, software architects are concerned with the analysis of fundamental system properties (like performance, reliability, ease-of-use, etc.) and the satisfaction of constraints (referring for instance to the use of particular databases or programming languages, or licensing issues) in order to choose the most adequate technologies to implement the functionalities of the system. Software architects want to play with some architectural model, understanding the consequences of an architecture over those properties, and exploring what-if questions (what if this response time requirement is relaxed? what if the programming language changes from Java to C#?), eventually giving feedback to analysts about feasibility of the analysis model.

They are not really interested in code but in more abstract questions, e.g. whether to use some enterprise pattern or other (e.g., Data Mapper [2] vs. Transaction Script [2]), which product better satisfies the needs implied by a pattern in the system (e.g., Hibernate[1] vs. .NET[2] DataMapper), to what extent database-related features (e.g., stored procedures, triggers, etc.) are preferable to writing code in some programming language, etc. Once an architecture model has been built, gene-rating code is a subordinated (although of course still very complex) task to perform.

The natural answer to this conflict is to explicitly distinguish the architecture model and the associated architecture design phase in the MDD transformation process. Therefore, we may reconcile both of the worlds: still we do have the advantages of MDD, whilst providing the possibility to software architects to take informed decisions according to system properties and constraints, building an architecture model from which code is derived in the rest of the MDD transformation process.

The goal of this paper is to present such a framework in the particular case of development of Information Systems (IS), which are one of the most typical types of software application nowadays (in different forms: client/server applications, Web-based systems, etc.). This type of systems are currently specified mostly using an object-oriented (OO) analysis methodology [3], and usually adopt a 3-layer architecture pattern [4], being the layers: presentation layer; domain, also named business, layer; and data, also named persistence, layer. In this scenario, if we refer again to the definition given in [1], we may say that:

- *Model of a system*. It is an OO analysis model that describes the functionality of the IS (i.e., what the system does) and a set of properties (e.g., efficiency, security, …) and constraints (e.g., which data base to use).
- *Real thing*: It is an architecture model generated from the analysis model, which basically declares which available technologies are used to structure the system and tackle the points behind the analysis model.
- *Transformation*. It is an assignment of elements from the functional part to the architecture, which satisfies the stated properties and constraints.

In this context, we have formulated the AR3L (Assignment of Responsibilities into a 3-Layered architecture model) framework. In AR3L, we take as functional specifica-tion a UML model [5] (although in fact other OO options like OMT [6] would be possible) and we describe both properties and constraints as a list of requirements in the form of restrictions stated over measurable factors. Concerning the architecture model, we focus on technology features (e.g., declaration of primary key in a data base schema, use of a combo box in a GUI, use of a dictionary data structure in the domain layer, etc.) and how they solve the functionality of the system. We consider this assignment, from functionality to technology features, as a simplified architecture model. In other words, the goal of this paper is to present a MDD framework that generates, from an OO analysis model, a 3-layer architecture model that satisfies a list of requirements about system properties and additional constraints.

Therefore, the transformation is a process that takes the following form (see fig. 1):

[1] http://www.hibernate.org
[2] http://msdn.microsoft.com/netframework

1. Infer the atomic elements that describe the system's functionality. Following the terminology introduced by [7], we call them *responsibilities*.
2. For each responsibility, select some technology feature that supports the stated requirements. These technology features are called *treatments* in the paper.
3. As a third step that is out of our current work, the transformation process would end by generating the source code from the architecture model.

In the first two steps, the architect constantly receives information from the transformation process and provides feedback, makes decisions, experiments, etc.

AR3L is a tooled framework. Among the several platforms that provide support for MDD (see [8, 9] for a survey), we have chosen AndroMDA[3] , an open source MDA [10] platform because it is highly customizable and it seems to be stable (in the sense of reliability and likelihood of existing over some time). As it is the usual case in this type of tools, AndroMDA is normally used for code generation but due to its customizability, we have been able to use it to create the architecture model as described above. With AndroMDA we have built a proof of concept of the AR3L framework that is also described in the paper.

The rest of the paper is organized as follows. In sections 2 and 3, we introduce the two processes that take part in the AR3L framework, responsibility identification and treatment assignment. In section 4, we give some details about the AndroMDA-based implementation. Last in section 5 we give the conclusions of our work.

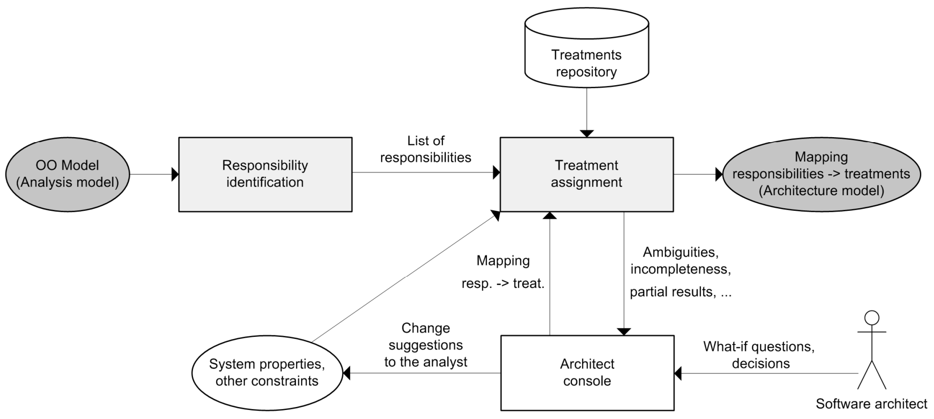

Treatments repository
OO Model (Analysis model)
Responsibility identification
List of responsibilities
Treatment assignment
Mapping responsibilities -> treatments (Architecture model)
Mapping resp. -> treat.
Ambiguities, incompleteness, partial results, ...
Change suggestions to the analyst
System properties, other constraints
Architect console
What-if questions, decisions
Software architect

Fig. 1. The AR3L framework.

2 AR3L: Responsibility Identification

In the context of OO analysis and design, the concept of responsibility was popularized by Wirfs-Brock at al. [7]. A precise definition of responsibility is given in [3]: "a responsibility embodies one or more of the purposes or obligations of an element". Responsibilities are identified in the analysis model.

[3] http://www.andromda.org

In our work, the analysis model has been declared to be a UML model. Therefore, the "elements" referred in the definition above are classes, attributes, operations, etc., and the responsibilities are obligations implied by these elements, e.g. being singleton (class), not exceeding some given value (attribute), satisfying some given precondition (operation), etc. More precisely, we have constructed a built-in taxonomy with almost 150 types of responsibilities, structured in a hierarchy of 6 levels. For instance, in the first level we have the categories: Class, Class Population, Attribute, Operation, Association, Association Population and Inheritance. Then, Operation is decomposed into Parameter, Precondition and Postcondition, etc.

Depending on the way they are declared, we may classify responsibilities into two main categories which hardly determine the way they will be identified in the model:

- *Graphical responsibilities*. They can be inferred from UML graphical elements, e.g. cardinality (of associations, attributes, etc.), properties (readonly, complete/disjoint, subset, etc.), dependencies (create, etc.) and by the like.
- *Non-graphical responsibilities*. They do not appear in the UML model, instead they are declared textually as integrity constraints expressed in natural language or more formal notations as the OCL. They may have a name. We distinguish two main subcategories:
  - *Permanent responsibilities*. They must be fulfilled in any valid state of the system, e.g. declaration of class identifiers or restrictions on values of class attributes. They are represented by means of class invariants.
  - *Event-driven responsibilities*. They must be satisfied when an operation is invoked. There are two types, preconditions and postconditions.

In the proof-of-concept we are describing in this paper, we have focused on one type of responsibility of each category and subcategory presented above, which we think are representative enough of the taxonomy. In particular, as graphical responsibility, we will use association cardinality; for permanent non-graphical responsibilities, declaration of class identifiers; for event-driven non-graphical responsibilities, a selected group of pre and postconditions (enumerated in section 2.3). We present these types in the rest of the section and we illustrate them using an example about a Conference Management System (see fig. 2). Table 1 presents the summary of all the responsibilities of those types found in the example.

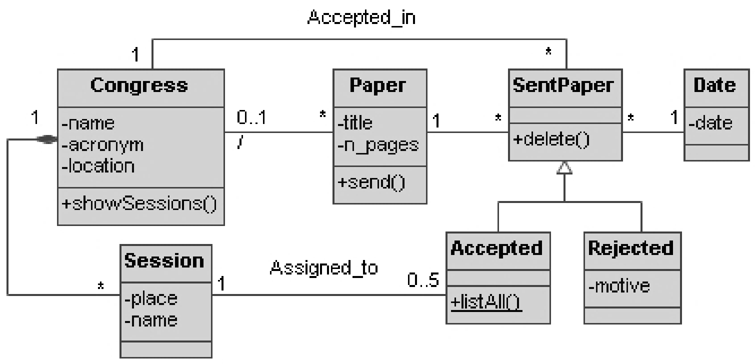


Fig. 2. Conceptual data model for the Conference Management System.

Table 1. List of responsibilities of the Conference Management System.

Responsibility	**Model Element Name**	**Responsibility description**
Cardinality	Paper	An object 'Paper' may have at most 1 object 'Congress' bound.
	SentPaper	An object 'SentPaper' must have exactly 1 object ‘Paper’ bound.
		An object 'SentPaper' must have exactly 1 object ‘Congress’ bound.
		An object 'SentPaper' must have exactly 1 object ‘Date’ bound.
	Session	An object 'Session' must have exactly 1 object ‘Congress’ bound.
		An object 'Session' may have at most 5 objects ‘Accepted’ bound.
	Accepted	An object 'Accepted' must have exactly 1 object ‘Session’ bound.
Identifier	Paper	Title is identifier.
	Congress	Name and acronym are both identifiers.
	Date	Date is identifier.
	Session	There cannot be two sessions with the same name in the same congress.
Pre / post	Paper.send	insertElement: post: Create a SentPaper.
	Congress.showSessions	listAll: post: Returns the set of Sessions bound to this Congress.
		existsElement: pre: The Congress must have some Session bound.
	SentPaper.delete	deleteElement: post: Delete a SentPaper.
	Accepted.listAll	listAll: post: Returns the set of all the Accepted Papers.
		notEmptyPopulation: pre: There must be some Accepted Paper

2.1 Association Cardinality Responsibility

Each association in the analysis model declares cardinality at each role. Given an association between A and B, the cardinality at one of its roles, let's say role at B, may define 0, 1 or 2 responsibilities depending on the form of that cardinality:

- Cardinality "*" means that there are not cardinality responsibilities induced.
- Cardinality of the form "0..n" induces a cardinality responsibility of upper bound (an instance of A may not be bound to more than n instances of B), whilst a cardinality of the form “n..*” induces a cardinality responsibility of lower bound (an instance of A may not be bound to more than n instances of B).
- Cardinality of the form "m..n" induces cardinality responsibilities of upper bound and lower bound.
- Cardinalities of the form "n" induce a cardinality responsibility of exact number (an instance of A must be bound to exactly n instances of B).

2.2 Identifier Responsibility

Most of the classes in the analysis model have a unique identifier (in addition to the OID) which allows distinguishing two different instances of a class. Identifier declaration is one of the most, if not the most, frequent types of integrity constraints that appears in specification models.

More precisely, a class C may have three types of identifiers:

- *Key*. The identifier is composed of one or more attributes declared in C.
- *Weak key*. The identifier is composed of one or more attributes declared in C together with one or more attributes that are identifiers of a class D such that a composition (in the sense of the UML) exists from C to D.
- *Alternative key*. In addition to a key, the class C may have some other attribute or set of attributes that uniquely identify C’s instances.

It is worth to remark that the three types of identifiers must satisfy several restrictions that are listed below:

R1 A class may have key or weak key, but not both at the same time.

R2 A class may not have more than one key or more than one weak key.

R3 For a class, the existence of alternative key demands the existence of a key.

R4 A class with weak key cannot be part of a composition of more than one key.

R5 For a class C with weak key such that D is the class linked by composition with C, it is required that D has either key or weak key.

R6 For a sequence of classes C1...Ck, such that a composition exists from Cj to Cj+1, $1 \leq j \leq k-1$, and each Cj has weak key, then Ck must have key.

In the example of fig. 2, the class Congress has both: a key, *name*, and an alternative key, *acronym* (note that deciding which identifier is key and which one alternative key is up to the analyst). On the other hand, the class Session has a weak key, composed by its attribute name and the key of the class Congress because Congress is a composition of Sessions, which means that two different sessions belonging to the same Congress may not have the same name.

2.3 Pre/Post Responsibilities

The behavior of the operations that appear in sequence diagrams is defined by means of a contract. Following Meyer's design by contract [11], we distinguish among preconditions and postconditions:

- *Precondition.* Condition that must be satisfied when executing an operation in order to get the agreed results. Typically this involves checking the value of one or more parameters and/or the state of the system.
- *Postcondition.* Describe an action that is performed during the execution of an operation. They may indicate a change in the state of the system or how a value is computed.

Due to the diversity of types of preconditions and postconditions, for the purposes of the proof-of-concept prototype we focus on a few types that are quite representative and allow us analyzing the actions to take in our approach. They are: insertion and deletion of elements (postconditions), get all the instances of a class (postcondition), check existence of an element (precondition), and check that a class has at least one instance (precondition). In Table 1, some examples appear covering all these cases for some operations introduced in the model: send a paper, show all the sessions of a conference, delete a paper sent and list all accepted papers.

3 AR3L: Treatment Assignment

Responsibility-driven design [7] may be seen as the process of assigning treatments (i.e., available technological features) to the responsibilities identified in the previous step. This view has been adopted more or less explicitly by the most widely used OO methodologies nowadays, both comprehensive (e.g., UP [12]) or more focused (e.g., Larman's [3], Fowler's [2], etc.).

3.1 Responsibilities, treatments and properties

In OO IS 3-layered design, a treatment may imply one or more layers. Component technologies such as Hibernate or EJBs[4], programming paradigms such as aspect-orientation, programming languages such as Java, graphical user interfaces or standards such as Swing[5] or XML, etc., determine the set of treatments available. Some examples on each layer are:

- *InputFilter* (presentation layer): ensures that an element exists using some GUI feature (e.g., a combo box).
- *Dictionary* (domain layer): in-memory structure that keeps track of the instances of a class.
- *OnCascade* (data layer): deletes elements using the OnCascade feature.

Fig. 3 presents a reference model of the AR3L framework. A treatment covers some of the responsibilities defined in our repository, e.g. InputFilter for the ExistsElement responsibility, or OnCascade for DeleteElement. Furthermore, each treatment has some effects on system properties and constraints, collectively known as requirements, e.g. Dictionary supports portability, whilst OnCascade damages it. The current responsibility assignment policy in AR3L binds treatments to responsibilities according to the values required on these requirements. To help the architect taking decisions, a treatment may have additional information bound (e.g., reports on technology, benchmarks, etc.) which can be consulted or updated during the responsibility assignment process. Technologies like Hibernate, Oracle[6], Swing, etc., implement treatments, therefore Oracle may implement OnCascade whilst Swing implements InputFilter. This information is important because of course the architect will generally not select two technologies of similar characteristics (e.g., two GUI libraries) to cover related responsibilities; also, it may be used as a guideline for the subsequent code generation phase. Some existing integrity constraints (e.g., coherence among responsibility and treatment hierarchies with respect to the is-covered-by association) complement this model.

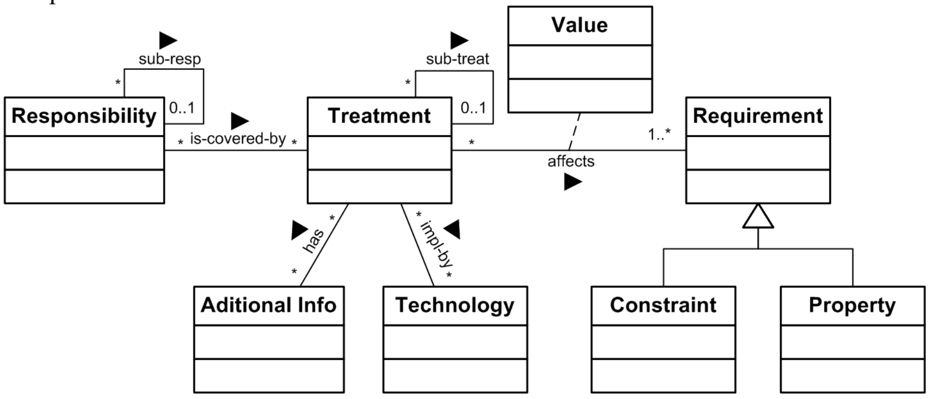


Fig. 3. Reference model of the AR3L framework.

[4] http://java.sun.com/products/ejb
[5] http://java.sun.com/javase/6/docs/technotes/guides/swing
[6] http://www.oracle.com

3.2 An Example: Application to the Conference Manager System

Next we provide scenarios that show the use of AR3L in different contexts. We present the scenarios and extract the information relevant to the responsibility assignment problem in the form of properties and constraints representing those scenarios. To do so, we need first to determine some treatments, properties and constraints to run the simulation.

For obtaining a representative sample in the 3-layer IS architecture context, we enumerate below some treatments bound to the different layers that will be considered in our Conference Management System example:

- *InputSizeControl* (presentation layer): ensures that a set of values (obtained via e.g. a combo box) does not exceed a given number of elements.
- *InputFilter* (presentation layer): ensures that an element exists using some GUI feature (e.g., a combo box).
- *CardinalityControl* (domain layer): piece of code that ensures that the role of an association does not exceed a given number of elements.
- *Dictionary* (domain layer): in-memory structure that keeps track of the instances of a class.
- *DataMapper* (domain and data management layers): provides isolation between the information in the database and memory (applying the Data Mapper pattern [2]).
- *SQL* (data management layer): allows direct manipulation of the data base.
- *Trigger* (data management layer): reacts when some condition is violated.
- *DBSchema* (data management layer): declares properties of tables.
- *OnCascade* (data management layer): deletes elements using OnCascade feature.
- *StoredProcedure* (data management layer): code at the data base.

Concerning properties and constraints we have chosen the following:

- Portability. To what extent an application is portable from one platform to others (Low, Medium, High).
- Operability. To what extent an application is easy to use. We will focus on one factor that heavily affects this criterion, complexity of user interface (Low, Medium, High).
- Environmental constraints. Which constraints exist about development that cannot be negotiated. As an example, we mention the data base (none, Oracle, PostgresSQL[7]).
- Development constraints. Which constraints exist about development that cannot be negotiated. As an example, we mention the development language (C++, Java, .NET).

Table 2 analyses the effect on these properties and constraints of the possible treatments for the association cardinality responsibility. We state for each treatment which are the values of properties and constraints that allow their application, for instance, to apply the InputSizeControl, there should not exist a requirement stating that Interface Complexity shall be Low. Empty cells mean that the treatment does not affect the property or constraint (e.g., using SQL does not affect Interface Complexity). The contents of the table reflect some heuristics (e.g., a treatment that

[7] http://www.postgresql.org

requires a data base has at least Medium technological dependency). Tables for the other responsibilities (not shown here for lack of space) are coherent.

Table 2. Influence of some treatments for association cardinality on system properties and constraints

Treatments for cardinality responsibility	User preferences											
	Portability			**Development language**			**Database**			**Interface complexity**		
	High	Medium	Low	C++	Java	.NET	None	Oracle	PostgreSQL	Low	Medium	High
InputSizeControl	**x**	√	√							**x**	√	√
DataMapper	**x**	**x**	√	**x**	√	√	**x**	√	√			
CardinalityControl	√	√	√	√	√	√						
SQL	**x**	√	√				**x**	√	√			
Trigger	**x**	**x**	√				**x**	√	√			
DBSchema	**x**	√	√				**x**	√	√			

We consider next three possible scenarios around the Conference Management System that will be under control of the software architect:

Scenario 1: we are interested in designing a subsystem that, given some keywords extracted from the title, obtains information about the papers and automatically generates the distribution of the papers into sessions. For modifiability and reusability purposes, this subsystem is designed as a service, with interface *assignPapers*(*allTitles*: Set(*String*)): Set(*nameSess*: *String* + *papers*: Set(*String*)).

The analyst concludes that: (1) the service does not need data base, it just performs some on-memory calculations; (2) interface complexity is Low, since both input (non-existing) and output (just a text file) are simple; (3) portability is not a big issue; (4) C++ seems to be the preferred language once checked the environment where the service will be deployed. Table 3, row 1, shows the result of this analysis. The architect receives the first feedback as shown in Table 4, Scenario 1 column. As a summary, all kind of responsibilities are assigned to the domain layer. The architect considers adequate the result and then confirms the proposal. Code production phase may start.

Scenario 2: after some time on production, the analyst is reported that the session assignment service behaves quite satisfactorily but not perfectly. Program chairs using the system require the ability to change manually the first assignment proposed by the system. Therefore, the analyst decides to modify the functionality of the system to allow changing paper assignment manually. As a consequence, the Complexity of User Interface is changed from Low to Medium (see table 3, row 2), to allow assignment of not much complex treatments that imply the presentation layer. Once the architect runs AR3L, he/she discovers some additional information bound to the treatment stating that presentation layer treatments are more difficult to implement using C++ than .NET. Since this decision was not a primary issue, he/she changes from C++ to .NET. Now, the assignment of treatments proposes presentation layer technologies as an alternative to domain layer for some responsibilities (see table 4, Scenario 2 column). The architect decides to exploit this possibility and informs AR3L about his/her choice to produce the final architecture model.

Scenario 3: again the intensive use of the service reveals that program chairs are still not happy, because not all the problems with session assignment are known in advance: authors may give preference about their travel dates later on, papers may be removed for several reasons, etc. Once informed, the architect decides that the solution is making the information about sessions persistent so it may be changed at any time. As an obvious option, he/she states as constraint that the data base shall be the same than the rest of the system, let's say Oracle. Once AR3L runs, the architect is informed that data layer treatments may not be exploited adequately unless portability is sacrificed. After some trade-off analysis, the architect decides to sacrifice portability (see table 3, row 3). As a result, under the agreement with the analyst, the architect gets a great deal of treatments applicable (see table 4, Scenario 3 column). Since DataMapper is a valid option for all responsibilities, it is finally chosen, and a technology implementing this treatment (e.g., Hibernate) may be selected.

In these scenarios we have followed a responsibility-driven treatment assignment process. However, other strategies are possible. For instance, architects usually feel more comfortable with some technologies that they know well, and may follow then a technology-driven treatment assignment process. For instance, Hibernate is a technology which currently has lot of success among IS developers. A possible strategy would be then to select Hibernate as starting point and then analyse which are the effects of this selection. If we assume this situation in Scenario 3, we may see that most of the responsibilities have then a default assignment to the DataMapper treatment (i.e., the treatment that is implemented by Hibernate). The architect would then confirm or not this treatment, perhaps changing to presentation layer treatments (i.e., InputFilter and InputSizeControl) when possible to avoid errors in lower layers.

Table 3. Mapping scenarios onto system properties and constraints

	Portability	Development language	Data Base	Interface complexity
Scenario 1	Medium	C++	None	Low
Scenario 2	Medium	.NET	None	Medium
Scenario 3	Low	.NET	Oracle	Medium

4 AR3L: An AndroMDA-based Tool

In this section we briefly describe the architecture of AR3L, which is built upon AndroMDA version 3. For space reasons, it is not possible to give the details of the solution, so we basically provide an overview being aware that full understanding would require more thorough explanations. We will focus to the most relevant aspect of using AndroMDA for our work, namely determining the information that must be included in the analysis model to facilitate responsibility identification.

Table 4. Assignment of treatments to responsibilities in the three scenarios

		Scenarios		
		Scenario 1	**Scenario 2**	**Scenario 3**
Responsibilities	**Identifier**	Dictionary	Dictionary	Dictionary
				SQL
				DBSchema
				DataMapper
	Cardinality	CardinalityControl	CardinalityControl	CardinalityControl
				InputSizeControl
				SQL
			InputSizeControl	Trigger
				DBSchema
				DataMapper
	existsElement	Dictionary	Dictionary	Dictionary
				InputFilter
			InputFilter	SQL
				DataMapper
	insertElement	Dictionary	Dictionary	Dictionary
				SQL
				StoredProcedure
				DataMapper
	deleteElement	Dictionary	Dictionary	Dictionary
				SQL
				StoredProcedure
				OnCascade
				DataMapper
	notEmptyPopulation	Dictionary	Dictionary	Dictionary
				SQL
				DataMapper
	listAll	Dictionary	Dictionary	Dictionary
				SQL
				DataMapper

4.1 AndroMDA

AndroMDA is an open source MDA framework that takes models of different types (usually UML models stored in XMI produced from case-tools) and processes them for obtaining some result of a given type, using a kind of plug-ins named cartridges. These cartridges implement M2T (model to text) transformations.

The AndroMDA components are managed by the AndroMDA core engine, which takes care of making all components of AndroMDA work together:

- The cartridges provide the ability to process UML standard models for a specific technology using template files.
- Template engines are components that generate the output of the process. In AndroMDA the most commonly used template engine is Velocity.

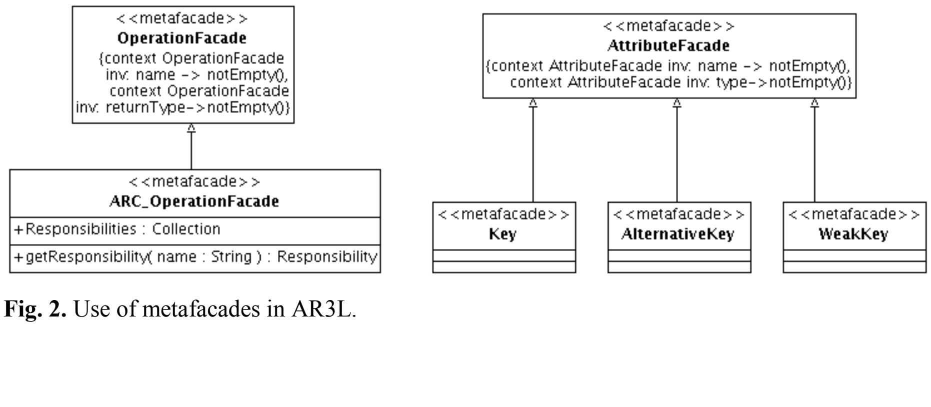
- AndroMDA Metafacades are facades that wrap the underlying metamodel implementation in such a way that AndroMDA designers are unaware of the details of that metamodel. Metamodels are MOF models such as UML 2.0, etc.
- Repositories allow switching out the repository that performs reading/loading of a MOF model. By default, AndroMDA reads models from XMI files.
- Translation libraries are used to translate OCL expressions into other languages. For example, a translation library could be used to translate an OCL body expression into a Hibernate-QL, EJB-QL OCL, Java, SQL, etc.

4.2 AR3L Tool: General View

As presented in the previous sections, general architecture of AR3L is divided into two parts, responsibility identification and treatment assignment. The first component uses the AndroMDA platform to interpret models and it has been implemented with a cartridge (the ARC cartridge). The technology assignment component has been implemented using standard Java and thus it could be eventually plugged into another different system.

To run this architecture, the analyst has to provide a UML model, the system properties and constraints, which are managed by the AndroMDA project system. The cartridge has access to this analysis model. In the process of detection, the cartridge builds as result a Data Transfer Object [2] packing all the detected responsibilities that is used in the next step. The assignment of treatments is done by a Velocity template that processes the responsibilities obtained from AndroMDA Core. The template calls to the treatment assignment module to assign a set of applicable technologies to each responsibility. In each call the template engine sends the user project information needed by the module. The result is a mapping from responsibilities to treatments.

Concerning the internal design of AR3L, there are two types of metafacades used in this project, one for responsibility management and another for stereotypes facilities. The metafacades used for managing and storing responsibilities also include the responsibility identification node, they extend the generic metafacades offered by AndroMDA. Each metafacade of this kind offers a collection of responsibilities and a method to access them easily (see fig. 4, operations facade). The second type of metafacades are just for convenience, they let us set the stereotypes related to keys and also to establish metamodel constraints (see fig. 4, attribute facade).



<<metafacade>>
OperationFacade
{context OperationFacade inv: name -> notEmpty(), context OperationFacade inv: returnType->notEmpty()}
<<metafacade>>
ARC_OperationFacade
+Responsibilities : Collection
+getResponsibility(name : String) : Responsibility
<<metafacade>>
AttributeFacade
{context AttributeFacade inv: name -> notEmpty(), context AttributeFacade inv: type->notEmpty()}
<<metafacade>>
Key
<<metafacade>>
AlternativeKey
<<metafacade>>
WeakKey

Fig. 2. Use of metafacades in AR3L.

4.3 AR3L Tool: Detailed View

In this section we describe in more detail how the three types of responsibilities detected in this paper are implemented in AndroMDA. According the internal architecture of AR3L, there are two issues to tackle in each type: how the responsibility is detected and how this detection is implemented.

4.3.1 Association cardinality responsibility

Association cardinalities are a typical example of graphical responsibility in which its identification is straightforward from the standard UML model, without needing additional elements, naming conventions or by the like.

Therefore, to implement the identification of this type of responsibility, we extend an AndroMDA metafacade; this extension will take care of the detection. At this point we face a situation that is recurrent in AndroMDA-based solutions, namely which metafacade is the most appropriate. In fact, each type of responsibility may be detected from several of the UML elements metafacade. Therefore, we need to establish a clear decision criterion in order to improve the understandability of the solution. Applying the same responsibility-driven design principle that we are advocating in the paper, we determine as general criterion to assign the type of responsibility to the metafacade of the UML element that is responsible of that type. This option is preferred in front of detecting the responsibility directly in the metafacade that corresponds to the UML element more related to that type of responsibility because this could provoke some ambiguity. Note that as result of this decision, the resulting code is more structured and more extensible: having such a clear organizational criterion facilitates extension because it is very clear which part of the system (i.e., metafacade) will have to take care of the new responsibility.

In the cardinality case, we basically have two options, to extend the metafacade of the UML element corresponding to associations or the one for classes. Applying the criterion above, we choose the metafacade for classes, which is ClassifierFacade, since each class is responsible for the roles that stem from it. Note that in this case, the first option would have been easier to implement, because for each association that exists in the model there would have been an instance of the metafacade taking care of the responsibilities inferred from that association's roles, that are exactly two (whilst in the case of a class, the number of roles is variable). Anyway, the chosen strategy is not difficult to implement, it just requires the following steps:

1. Building a metafacade that extends the functionality of the generic Classifier Facade. We remark that if other types of responsibilities also are assigned to UML classifiers, this same extension will be used for them (as we show later).
2. Implementing the detection code inside this new metafacade. This code just analyses all the associations of the class and collects all the responsibilities generated by the cardinalities as specified in the beginning of this section.
3. Developing a Data Transfer Object (DTO [2]) to represent this responsibility. Using DTOs makes our solution less dependent on the technology.

4.3.2 Identifier Responsibility

The consideration of non-graphical responsibilities raises the fundamental problem of identifying these responsibilities in the specification model. Several strategies are possible: using UML elements such as notes, stereotypes and tagged values; using naming conventions; etc.; in fact, several strategies could be eventually used for different types of responsibilities. To select in each case the detection strategy we have take into account the following values: analyst effort, understandability, expressiveness, standardization and solution implementation effort.

For the identifier case, we have selected attribute level stereotypes strategy. Each attribute that is part of an identifier is preceded by the appropriate stereotype. Note that this information must be provided by the specifier. Concerning implementation of the strategy, the first decision is to assign the responsibility to the UML element class again, since identifiers are a concept bound to classes. Therefore, we use the extension of ClassifierFacade presented in section 3.2. Then, we make the following steps:

1. Extending the AttributeFacade for creating one new facade for each of the three types of stereotypes.
2. Including in the ClassifierFacade extension the restrictions that keys must fulfill (R1-R6 at subsection 2.2) in the form of OCL expressions that will be executed by the AndroMDA Core. There are two exceptions. First, R2 is ensured by construction, since all the attributes stereotyped as Key or WeakKey are considered to be part of just one identifier. Second, R6 is a recursive expression difficult to express in OCL and then it will be treated ad hoc.
3. Implementing the detection code inside the extended metafacade. In this code we look in each class attribute and compare it with the previous mentioned stereotypes to compose the keys. For weak keys, this is done in a recursive way.
4. Implementing the code for the restriction R6. This code is included in the same metafacade in the validation operation invoked by AndroMDA when the specification model to process is validated.

4.3.3 Pre/post Responsibility

In the case of pre/post we have decided to use naming conventions in the textual description, because adding new subtypes of responsibilities means just recognizing a new name. The names used are insertElement, deleteElement, listAll, existsElement and notEmptyPopulation.

The implementation consists of the following steps:

1. Extending the OperationFacade for creating a new facade that may be used later on for dealing with other types of responsibilities assigned to operations.
2. Implementing the detection code inside the extended metafacade. This code gets all the constraints of an operation and compares the element name with the agreed pre/post names.
3. Developing DTOs to represent each of the pre/post responsibilities.

It is worth to remark that in all the cases, when a metafacade is extended, it is necessary to process the new model using Maven to update those Java files that are generated automatically by AndroMDA. Furthermore, it is necessary to modify an XML file (metafacades.xml) to make the new metafacades usable.

5 Conclusions and Future Work

In this paper we have presented AR3L, an AndroMDA-based system for assigning treatments to responsibilities extracted from an analysis model (which takes the form of a UML model), according to some system properties and constraints, in the context of 3-layer-architecture information systems. This assignment is considered as a simplified architecture model, since it reflects which are the technologies (treatments) that compose the architectural solution of the system.

We consider that the most important contribution of our work is to reconcile classical MDD engineering with the important role that software architects play in software engineering. Software architects have lots of knowledge, skills and expertise that are very difficult to include fully in the MDD transformation process. Furthermore, this knowledge, skills and expertise depend on a lot of factors that are difficult to quantify (e.g., information about performance requires lots of benchmarking to be reliable) or to represent (e.g., how to represent information about company's political or strategic issues?) in a software engine. The need for this kind of mixture between pure MDD and human participation is recognized by other authors, e.g. [13]. There are few proposals that tackle non-functional requirements in the MDD process, e.g. [14], that one is centered on usability evaluation for building user interfaces.

Another contribution has been to use an existing technology like AndroMDA as underlying platform for our prototype. As a result, we have got a system architecture that is highly extensible (with respect to management of responsibilities, system properties and constraints), maintainable (easy to replace some components by others for evolvability or customization), and reasonably portable (the dependency on AndroMDA has been kept to the minimum extent, basically to the part of converting a model responsibility into some invocation to a metafacade operation). It is worth mentioning also that AndroMDA community is very lively (in fact, they recently released AndroMDA 4.0 for public review) which is clearly a point supporting evolvability of our AR3L tool.

Concerning general applicability of our proof-of-concept, we think that the concepts can be abstracted to a wider context and we are already working on this line; concretely we are building a more generic framework called RDT [15], but of course scalability is an open issue that should be experimented soon. The taxonomies that exist in the AR3L reference model may help to tackle this issue, both to incorporate new technologies to the framework as they appear, to establish their effects on properties and constraints, and to use it by working mostly at higher-levels of the hierarchy. On the one hand, both assignments and responsibilities are hierarchically structured, leveraging the complexity of understanding the repositories of these types of elements. On the other hand, hierarchies may be useful for propagation: if some treatment affects a requirement, its ancestors will also affect it; and if some treatment is implemented by a technology, its heirs will too. This default behaviour may be broken e.g., a technology may not implement a particular feature that one could expect. Another point worth mentioning is the use of UML for the functional part of the analysis model, instead of ad hoc domain specific languages, we think that it may make more attractive our approach to the IS development community which may still use their own modeling language.

Future work embraces basically two lines of action. On the one hand, to evolve our current proof-of-concept into a more complete prototype, by developing some case studies more complex that the example shown in this paper. On the other hand, to deeply explore the relationship among the software architect and the MDD transformation process. For instance: in this paper we have proposed a fully automatic responsibility identification process. This has two possible drawbacks. Firstly, as shown in the paper, analysis model must carry more information (stereotypes, naming conventions, etc.) to allow fully automatic identification, and it must be assessed whether this extra modeling effort (which furthermore does not allow importing arbitrary UML models) is worthwhile. Secondly, some decisions may not be known in modeling time, e.g. navigation of associations: a non-navigable role must not generate any association cardinality responsibility.

You may download the AR3L tool implementing the proof-of-concept at www.lsi.upc.edu/~gessi/AR3L.

References

1. S.J. Mellor, A.N. Clark, T. Futagami: “Model-Driven Development”, IEEE Software, vol. 20, no. 5, pp. 14-18, September 2003.
2. M. Fowler: “Patterns of Enterprise Application Architecture”, Addison-Wesley, 2003.
3. C. Larman: “Applying UML and Patterns (3rd edition)”, Prentice Hall, 2004.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad: “Pattern Oriented Software Arquitecture”, Vol. 1, John Wiley, 1996.
5. J. Rumbaugh, I. Jacobson, G. Booch: “The Unified Modelling Language Reference Manual (2nd edition)”, Addison-Wesley, 2004.
6. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: “Object-Oriented Modeling and Design, Prentice Hall”, ISBN 0-13-629841-9
7. R. Wirfs-Brock, B. Wilkerson, L. Wiener: “Designing Object-Oriented Software”, Prentice Hall, 1990.
8. N.A. Tariq, N. Akhter: “Comparison of Model Driven Architecture (MDA) Based Tools”, In Proceedings 13th Nordic Baltic Conference (NBC), 2005.
9. J. Cabot, E. Teniente: “Constraint Support in MDA Tools: A Survey”, In Proceedings 2nd European Conference on Model Driven Architecture Foundations and Applications (ECMDA), LNCS 4066, Springer-Verlag, 2006.
10. OMG: “Model Driven Architecture (MDA)”, Object Management Group, ormsc/2001-07-01, July 2001.
11. B. Meyer: “Object-Oriented Software Construction”, Prentice Hall, 1997
12. I. Jacobson, G. Booch, J. Rumbaugh: “The Unified Software Development Process”, Addison-Wesley, 1999.
13. B. Humm, U. Schreier, J. Siedersleben: “Model-Driven Development – Hot Spots in Business Information Systems”, ECMDA 2005
14. S. Abrahao, E. Insfran: “Early Usability Evaluation in Model Driven Architecture Environments”, Sixth international Conference on Quality Software, 2006.
15. D. Ameller, X. Franch: “Assigning Treatments to Responsibilities in Software Architectures”, Euromicro'07, 2007.