A Method for Establishing Information System Design Practice

Dalibor Krleža
Global Business Services
IBM
Miramarska 23, Zagreb, Croatia
dalibor.krleza@hr.ibm.com

Krešimir Fertalj
Department of Applied Computing
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3, Zagreb, Croatia
kresimir.fertalj@fer.hr

Abstract—**Information system design and development practice did not evolve much in the last decade. Methodologies for design and development of information systems are still separating activities for design and development. Design is always done prior to the development, resulting in deliverables that cannot be reused in the development process. The deliverables of the design process are used by developers only as a blueprint of the information system. The Model Driven Architecture promised to change that by introducing model transformations. The whole idea introduced in the Model Driven Architecture raised the question of model quality. It is not possible to have a correct and complete transformation if models of the information system are not of high quality. It is very hard to achieve a sufficient level of model quality on a big project. The size of a project team makes it hard to control all contributions from designers, ensuring that these contributions comply with the project design practice. In this article, we deal with these issues by providing a method that allows control of the information system design process. This method provides guidance to designers in the project team by offering selection of patterns and transformations that are applicable to the current state of the information system design. The library of patterns and transformations represents previous design and development practice, containing knowledge developed during previous projects. The method proposed in this article allows selection of patterns and transformations that are suitable for the project, constraining and guiding the contributions of designers.**

Keywords-modeling; guidance; design; pattern; transformation.

I. INTRODUCTION

The practice of information design and development still has a number of issues that needs to be addressed. Current information system design and development practice is still mainly manual, and uses design mostly for documentation purposes. In order to improve the ratio of successful and unsuccessful projects, as well as to cut down the costs of the projects, more effort needs to be put into defining better and efficient design practices that can improve project team coordination and communication, as well as traceability and quality of the project deliverables.

This article is an extension of work done in [1]. Although the focus of this article is mainly on design practice, code development is tackled and mentioned as the result of the design process. The definition of a design project and the design process used in this article is given by Ralph and Wand [2] in the form of a conceptual model. The conceptual model includes a very precise definition of terms "design project", "knowledge", and "practice". Ralph and Wand argue that design is more important than code development, simply because design elements are better than code for communicating the rationale for structural and behavioral decisions. By giving potential applications of their conceptual model, Ralph and Wand set two challenges considered in this article:

1. Design knowledge management system - A system for storing and managing the design knowledge.
2. Design approach classification framework - A framework that enables classifying of design approaches. Such framework must give guidance for selection of a design approach and comparative research on different approaches to designers.

The proposed method is mainly based on the Model Driven Architecture (MDA), standardized by the Object Management Group (OMG) [3]. The MDA is an information system design approach based on models and model transformations. Using the MDA, an information system is designed (and developed) through several abstraction levels, from business oriented models to technically oriented models: Computational Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). The process of designing includes transformation of models between different levels of abstraction. Eventually, PSM is transformed into code. The promise of the MDA approach is to reduce the time and effort needed to code an information system, by refocusing on delivering meaningful details in the design activities.

However, the MDA is not perfect, and has its own issues. Gholami and Ramsin [4] are giving Strengths, Weaknesses, Opportunities, and Threats (SWOT) analysis of the MDA. Some of the issues recognized in this analysis are addressed by the proposed method:

1. Need for creation of custom transformations consumes a lot of time. Resolution of this issue must be in establishing reusable design practices.
2. Model quality issues. Without models of adequate quality, transformations cannot be successfully used and applied.

Methodologies for design and development of information systems are blueprints for processes [5] that allow a project team an organized way of designing and developing of an information system. Relying only on methodologies for design and development of information systems is not necessarily producing a model of high quality, because many of these methodologies do not incorporate design practices. For reference, the quality model given by Lange and Chaudron [6] is used. There are MDA specific methodologies that are addressing some of the MDA issues. Chitforoush, Yazdandoost, and Ramsin [7] give an overview of such MDA specific methodologies. Most of these methodologies were developed for specific projects, having built-in design practices that do not allow flexibility when needed. Some generic design and development methodologies, such as Rational Unified Process (RUP) [8][9], also rely on model based design.

The absence of the design practices can result with a model of poor quality, i.e., the model is untraceable, hard to transform and hard to analyze. One way to solve these problems is to establish design practices for the project. Established design practices must ensure that models are uniform and of high quality. According to the quality model [6], this means that all models are traceable, complete, and consistent, and that models correspond to the information system being designed and developed.

In this article, a method for establishing and imposing design practices is proposed. In the context of the MDA, establishing design practices means defining and imposing of patterns and transformations that need to be used during the design process of the information system. Reusing successful patterns and transformations from previous projects can help to establish design practices. The proposed method extends methodologies for design and development of information systems by utilizing existing OMG specifications to achieve guidance in the design process that addresses some of the MDA issues [4], and answers challenges set by Ralph and Wand [2]. The proposed method is an add-on to existing design and development methodologies. A certain level of compatibility between a design and development methodology and the proposed method is needed. Some of the MDA methodologies might be incompatible with the proposed method, since they already contain design practices. Tools used for designing and developing of an information system must have features that allow a project team to follow the method proposed in this article.

In Section II, a modeling space is defined. The modeling space allows combining all models of an information system together, providing relationship between them, and defining their purpose. In the same section, a relationship between pattern instances and models of different abstraction levels is given. Section II also includes the definition of a modeling library that contains design practice from previous projects. In Section III, current design practice in the context of generic methodologies is discussed, which helps understand how pattern instances are created during the course of the project. In Section IV, an overview of the pattern instance transformation is given. The pattern instance transformation is essential for the method proposed in this article. In Section V, the tracing and transformation language is defined. This language is used to bind pattern instances together, and help to establish tracing between model elements. In Section VI, an overview of the method for establishing the design practice is given. Section VII contains an example that presents how the proposed method works on a real life scenario.

II. Modeling Space

The proposed method deals with all models involved in the project. According to the MDA specification [3], "model transformation is the process of converting one model to another model". From a transformation point of view, the MDA deals with models that are directly involved in a transformation. This can involve at least one model. However, the transformation does not need to include all models involved in the project. From a design and development methodology point of view all models are somehow connected. All models that are part of the project need to be accessible by a tool that implements the proposed method. Many design and development tools use containers for keeping models and model elements [10] together. More than one container can be used for the project. Therefore, the tool itself must have the ability to keep relationships between modeling elements placed in different containers. The proposed method must deal with all modeling elements of the project, no matter how many containers are there. The conclusion is that all models and model elements must be observed as a part of one big modeling space.

A modeling space is a notation that can be used to represent the classification of all models that are part of the project. The modeling space can be drawn as a square box containing all possible models of a designed information system. The modeling space must follow the MDA philosophy, i.e., support different levels of abstraction given in the MDA specification.

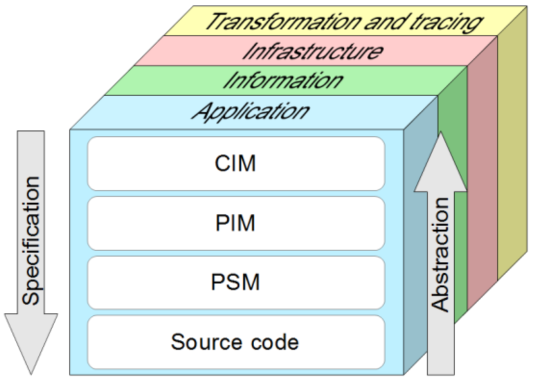


Figure 1. Structure of a modeling space

The modeling space presented in Figure 1 contains different layers, representing respective aspects or viewpoints of the designed information system. The modeling space contains four layers. The application layer is comprised of models with the business logic. The information layer is comprised of information and data models. Models containing architecture details and infrastructure nodes are placed in the infrastructure layer. Finally, there needs to be a specific layer for transformation and tracing models. Of course, a number of layers and their purpose depend on a set of models representing an information system design. One model can belong to multiple layers. For example, a model containing requirements can easily be considered for application, information, and infrastructure related, since requirements are determining all aspects of the future information system. The modeling space must also support a clear distinction between abstract and detailed models. Abstract and computing independent models are placed on top of each layer. Models with more details are closer to the bottom of the layer.

Each model is a set of model elements. These elements originate from a modeling language, such as UML [10]. A set of models together represent the design of an information system. However, there are sets of model elements in every model that are meaningful for designers. These sets of model elements, or patterns, can be seen as reusable solutions to problems. Every level of abstraction can have its own repeating patterns of model elements. For example, CIM can contain repeating sets of model elements that can be interpreted as requirements or business processes, PIM can contain use cases or components, and PSM can contain implementation of components defined in PIM.

CIM patterns are usually created early in the project, and they depend on used architecture as well as how business analysis is performed. These high level abstract patterns have the biggest impact on the design of an information system. PIM patterns are derived from architecture and computational independent patterns. They represent an elaboration of CIM patterns within an architectural context. The most detailed are PSM patterns that represent the implementation of PIM patterns for a specific infrastructure yielded by the previously determined architecture.

A. *Modeling library*

In order to establish the proposed method, a library of modeling patterns and transformations must be established. The usual way to create a pattern library is by using a template document [11]. An example of online accessible pattern library can be found on [12]. This library is created by using Cloud related pattern language defined by Fehling et al. [13]. Gamma, Helm, Johnson, and Vlissides [14] propose the pattern library of basic object-oriented patterns, visualized in the UML. Hohpe and Woolf [15] propose the enterprise integration pattern library.

However, previously mentioned pattern libraries are not suitable for use by the proposed method, since solutions in these libraries are not structured, and cannot be browsed directly by a tool that implements the proposed method. In this article the Meta Object Facility (MOF) [16] family of modeling languages is used. MOF is a metalanguage standardized by the OMG. A pattern library suitable for the proposed method must utilize MOF based repository for the solution of a pattern. A good description of MOF based repository is given by Frankel [17].

The proposed modeling library can be used as the design knowledge system presented in [2]. The modeling library must have all needed features for storing and managing patterns and transformations that constitute the design knowledge.

Collecting modeling patterns can be done from existing pattern libraries, or models of already developed information systems in previous projects. Then, these patterns are inserted into the modeling library suitable for the proposed method. Collection from existing models can be done manually or automatically by detecting repetitions. Detection itself can be done by the graph matching method [18]. Pham et al. [19] propose the graph matching method for detection of cloned fragments in graph based models. According to their definition, repetitive fragments that are similar enough can be considered for clones or patterns. A similar approach can be applied to UML models.

Falkenthal, Barzen, Breitenbücher, Fehling, and Leymann [20], argue that concrete solutions are lost in the process of pattern writing. The reason for that is the need for discarding some of the solution details. If a pattern is created by using already existing information system design, then discarded model elements are the ones that need to be contributed by a designer through the process of pattern elaboration.

A pattern is a class, a blueprint that binds one or more model elements together. Application of a pattern means instantiation [20][21] within at least one model in the modeling space. Applying the pattern does not mean that the modeling is completed. Adding details and further elaboration of the pattern instance is needed, to bridge the gap between the selected pattern and final solution that was lost in the pattern writing process, which is environment and context dependent.

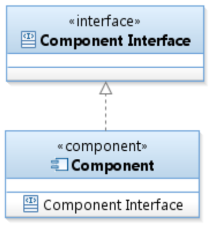


Figure 2. Example of a simple modeling pattern: a component and an interface

Figure 2 represents a pattern that is comprised of an empty interface and a component. After applying this pattern a pattern instance is created. Further elaboration of the pattern instance must add interface details, operations and parameters, subcomponents, and additional interfaces.

III. Mapping between Methodology and Pattern Sequence

Porter, Coplien, and Winn [22] have shown that pattern sequences are important, i.e., aggregation and composition of patterns rely on the order how they are applied. A pattern library is just a set of patterns that have particular relationships among them. Even with related patterns, a slightly different order of pattern application can have different results.

The novelty introduced by the proposed method is a way to define possible pattern sequences through transformations. Transformations inside the previously defined modeling library have purpose to determine relationships between patterns, and to introduce the possibility to transform pattern instances based on patterns from the modeling library. This way, sequences are determined by transformations that are part of the modeling library.

In this article, we also argue that the selected design and development methodology (RUP for example) significantly contributes to the pattern sequence. The selected methodology defines high-level phases of a pattern sequence by providing the order how models are created and elaborated. There is a correlation between the set of transformations in the modeling library and high-level pattern sequence driven by the design and development methodology. The modeling library must contain all needed transformations that allow this high-level sequence to be completed according to the methodology.

A pattern sequence has fine course within a single project task. This fine course, or low-level sequence, is a set of activities within the task needed to complete a model, or a set of models. Transformations in the modeling library must also support these low-level sequences.

A. Methodology driven, high-level pattern sequence

CIMs are usually created very early in the project. In the RUP, business models are created in the Inception phase. It means that selecting and applying CIM related patterns, as well as further elaboration, can be done very early in the project. These patterns can be classified as functional requirements, non-functional requirements, business processes, or business use cases. The idea is to have these patterns and related transformations ready for use in the modeling library. Elaboration of newly created pattern instances in CIMs can be done in the Inception phase.

PIMs, part of the PSMs, architecture models, and infrastructure models, are created in the Elaboration phase. In this phase, we do most of an information system design, and take the most important decisions. In the Elaboration phase, patterns used in CIMs provide guidance for choosing patterns that could be used next. For example, usual patterns that could be used here contain use cases, components, and nodes.

The PSM is usually the last step in the design of an information system. The ultimate goal is to get the source code and deployment units. Therefore, the PSM must contain pattern instances that define a sufficient level of details for transformation into the source code, in a way that there is less work as possible for developers. Pattern instances in the PSM are mostly implementation of pattern instances in the PIM. For example, in the Component-Based Design (CBD) [23], the PSM contains platform specific implementations of components defined in the PIM.

Figure 3 provides a visual course of a project, high-level sequence of work on models and low-level sequence of pattern instantiation, transformation, and elaboration. Generally, as the project advances through the phases defined in the RUP, models become more and more specified and elaborated, until the level of actual program code. For simplicity, only one pattern instance per model is used. Models are represented by circles marked as Mi, pattern instances are represented by circles marked as Pj and transformations are represented by edges marked as t_k. Figure 3 represents an example with the following detailed low-level sequence:

1. The pattern instance P_1 is created, containing a business process. This pattern instance can be done using BPMN [24].
2. The pattern instance P_2 is created, containing a set of model elements that represents architectural decisions about selected middleware (application server, database). UML [10] can be used for this purpose.
3. Transformation t_1 is used to extract a business object from the business process information flow into the

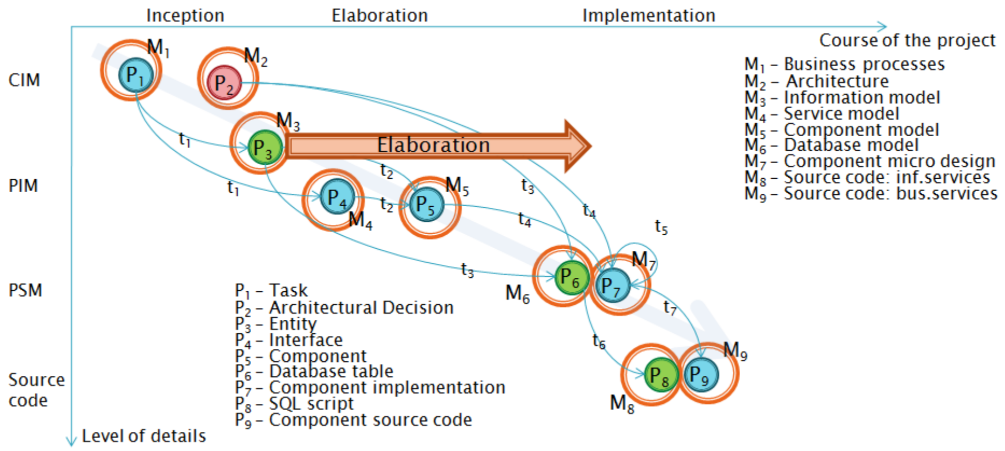


Figure 3. RUP and advancement through the design of an information system

resulting pattern instance P_3 that represents an entity of the information system. Transformation t_1 is also used to extract the interface from a task that belongs to the business process in the pattern instance P_1. As the result of transformation t_1, the pattern instance P_4 is created, representing the interface of the source task.

4. Pattern instances P_3 and P_4 are elaborated by adding operations and attributes.
5. Transformation t_2 is used to create the pattern instance P_5 from pattern instances P_3 and P_4. The pattern instance P_5 contains a component [10][23] that has an association to the entity in the pattern instance P_3, and realizes the interface in the pattern instance P_4.
6. Transformation t_3 is used to create the pattern instance P_6 from pattern instances P_2 and P_3. The pattern instance P_6 represents a table that reflects the entity in the pattern instance P_3. Transformation t_4 must take in account modeling element that contains the architectural decision about used database.
7. Transformation t_4 is used to create the pattern instance P_7 from pattern instances P_2 and P_5. The pattern instance P_7 represents implementation details of the component in the pattern instance P_5, taking into account the architectural decision about used application server.
8. Optionally, transformation t_5 can be used in case when additional standard implementation is added to the implementation of the component in the pattern instance P_7.
9. Transformation t_6 is used to create a Structured Query Language (SQL) script that can be used to create the table from the pattern instance P_6.
10. Transformation t_7 is used to create the Java code for the EJB defined in the pattern instance P_7.

As the design of an information system advances through the project, designers can create new pattern instances, or elaborate on existing ones. A new pattern instance can be created to document business need, reflect already existing functionality that can be reused, or by transforming from already existing pattern instance in the modeling space. Transformation between pattern instances is probably the most used option. Elaboration of the existing pattern instances is also very important. Once a new pattern instance has been created, it must be elaborated in subsequent project activities.

IV. Pattern instance transformation

Model transformation is a key procedure in the MDA. The MDA specification [3] contains various different model-to-model transformation combinations and examples. Transformation can be done within the same model, between two different models, for model aggregation, or model separation. Grunske et al. [25] are presenting an important notion of "horizontal" and "vertical" transformations. Horizontal transformation is done between models of the same abstraction level. Typical horizontal transformation is PIM to PIM, or PSM to PSM. Any transformation within the same model is also a horizontal transformation. Vertical transformation is done between models of different abstraction levels, or from a model to the source code. A transformation from PIM to PSM, or from PSM to the source code is vertical transformation.

Model transformation can be done either manually or automatically. Manual model transformation is more common than we think. It is not unusual for a designer to start modeling from scratch by using models delivered earlier in the project. When a modeling language is structured and formal enough, automatic transformation can be used. All modeling languages derived from MOF can be transformed automatically.

Automatic transformation takes elements of a source model and converts them into elements of a target model by using transformation mapping. Transformation can be additionally used to establish relationships between models, or to check consistency of elements between source and target model. Czarnecki and Helsen [26] elaborate a number of model transformation approaches. Most used are graph based transformations and transformation languages. Transformation languages can be declarative or imperative. The OMG standardized group of MOF based transformation languages named Query/View/Transformation (QVT) [27]. QVT Relational language (QVT-R) is a typical example of a declarative approach with the graphical notation. QVT Operational language (QVT-O) is an example of an imperative approach. In this article, we are using the declarative approach.

In order to understand which transformation features are needed for the method proposed in this article, basic principles of a model transformation must be observed. Let us define a modeling space as a finite set of models

$$M_S = \{M_1, M_2, \dots, M_n\} \tag{1}$$

Each model is a finite set of elements

$$M_i = \{e_1, e_2, \dots, e_m\} \tag{2}$$

A transformation is a function

$$tr: M_S \rightarrow M_S \tag{3}$$

that takes a set of elements er_{So} from a set of source models $So \subseteq M_S$ such that $er_{So} \subseteq \bigcup So$, and translates them into another set of elements er_{Ta} in a set of target models $Ta \subseteq M_S$, such that $er_{Ta} \subseteq \bigcup Ta$. A transformation can be done within the same model $So = Ta = M_i$, or between two disjunctive sets of models $So \neq Ta$. Since a transformation can have multiple models from source and target side, these sets do not need to be disjunctive $So \cap Ta \neq \emptyset$, meaning that the transformation can include the same model M_i on source and target side, or $M_i \in So \land M_i \in Ta$. A transformation can use the same source and target elements, meaning that

$er_{So} \cap er_{Ta} \neq \emptyset$ when $So \cap Ta \neq \emptyset$, or it can use two disjunctive sets of elements $er_{So} \cap er_{Ta} = \emptyset$.

From a pattern point of view, each pattern instance is a set of model elements. This definition is valid for cross model pattern instances as well. All pattern instances in the modeling space M_S form a finite set of pattern instances

$$M_P = \{p_i : 0 < i \leq m \wedge p_i \subseteq \bigcup M_S\} \tag{4}$$

In this context, transformation is a function

$$tr: M_P \rightarrow M_P \tag{5}$$

Such transformation takes a set of source pattern instances $p_{So} \subseteq M_p$, and translates them into model elements that form a set of target pattern instances $p_{Ta} \subseteq M_p$. More precisely, transformation can be written as

$$tr: p_{So} \rightarrow p_{Ta} \tag{6}$$

Every transformation can be encapsulated in a black box implementation. Such an approach is used in [27] along with the QVT specification. According to the QVT specification, every transformation can be defined as a black box having an interface that depends on the context of transformation usage.

A. Transformation rules

Using a declarative approach for transformation of pattern instances means that every transformation can be represented as a set of transformation rules that define the relationship between a set of source model elements and a set of target model elements [26][27][28].

Czarnecki and Helsen [26] are giving important features of a declarative transformation consisting of transformation rules. As defined in [26], each transformation rule has "the left-hand side (LHS) that accesses the source model and the right-hand side (RHS) that expands in the target model". In this article, LHS is referred as "the source side" and RHS as "the target side".

Jouault and Kurtev in [29] are defining execution model for ATLAS Transformation Language (ATL) rules. Matching transformation rules in their model have a declarative part and an optional imperative part. The execution algorithm is matching the declarative part of a transformation rule, which is then fully executed (the declarative and the imperative part) in case that the transformation rule matches the supplied source pattern. It is very important to notice that declarative transformation rules are independent of each other, and that the execution algorithm does not guarantee the order of execution.

Transformation and related transformation rules, especially if they are written in a declarative way, are logic programs [30]. Transformation tr (6) can be defined as a logic program comprised of a set of rules

$$tr = \{r_1(p_{So}), r_2(p_{So}), \dots, r_n(p_{So})\} \tag{7}$$

The conditional part of every rule in previously defined set is comprised of atomic logic functions that involve model elements

$$r_i(p_{So}) \leftarrow a_1(y_1, p_{So}) \wedge a_2(y_2, p_{So}) \wedge \dots a_m(y_m, p_{So}) \tag{8}$$

where $y_j \in p_{So}$ is a model element in source pattern instances of the transformation tr. According to (8), rule r_i is matched only if all of the atoms are evaluated as true.

A transformation written in the QVT-R has two different modes: checking mode and enforcement mode. In the checking mode, transformation rules can be used to validate correctness and completeness of involved pattern instances. In the enforcement mode, transformation rules can be used for creating, updating, or deleting model elements in target pattern instances, in order to reflect all the details found in source pattern instances.

1) Applying transformation

As already defined, a transformation takes a set of modeling space model elements and translates them into another set of model elements. Earlier definition (6) shows that the transformation can include pattern instances as model element containers.

According to (8), every transformation rule consists of a set of atoms that are used to determine whether model elements in a set of source pattern instances are matching conditions of the transformation rule or not. So far, there are no additional conditions in (7) that would indicate whether a transformation can be applied to the source pattern instances or not. In order to define conditions whether a transformation can be applied or not, a set of transformations is divided on two disjunctive subsets. We define a set of "mandatory transformation rules", which need to match the source pattern instances for transformation to be applicable.

$$mtr(tr) = \{mr_1(p_{So}), mr_2(p_{So}), \dots, mr_n(p_{So})\} \subseteq tr \tag{9}$$

When a transformation is applied, is it certain that all action parts of mandatory transformation rules will be executed, if all mandatory transformation rules match supplied source pattern instances, i.e., transformation is applicable to this set of pattern instances. We also define a set of "optional transformation rules", which do not need to match the source pattern instances for transformation to be applicable.

It is possible that an atom in a transformation rule of the applied transformation matches more than one model element in the source pattern instances. A tool implementing the proposed method must allow a designer to choose which model element will be transformed. For example, a model can contain a set of use cases. The designer applies a generic transformation that can be applied to any use case. Obviously, the tool must allow him to choose which use case will be transformed by the applied transformation.

$$otr(tr) = \{or_1(p_{So}), or_2(p_{So}), \dots, or_m(p_{So})\} \subseteq tr \quad (10)$$

If conditional part of an optional transformation rule does not match supplied set of source patterns, the action part of this rule is not executed. Optional transformation rules can be used to transform elaborated details.

This might lead to a conclusion that mandatory transformation rules need to cover transformation of model elements that comprise a pattern in the pattern library, and that optional transformation rules must cover transformation of all model elements added after a pattern instance was created, i.e., elaborated model elements. While this is generally correct, mandatory transformation rules might include some elaborated details, making this transformation applicable only after elaboration of the pattern instance. This way, a designer is forced to contribute details before proceeding further in a pattern sequence.

When a transformation is applied, execution of the transformation must perform several different steps.

As the first step, the set of mandatory transformation rules of the applied transformation must be matched with the supplied source pattern instances. If all mandatory transformation rules are matched on the source side then the transformation can be applied to the supplied source, i.e., the transformation can be applied to the source pattern instances that contains all model elements needed by the mandatory transformation rules. Transformation applicability can be expressed as

$$A(mtr(tr), p_{So}) \leftarrow mr_1(p_{So}) \wedge \dots \wedge mr_n(p_{So}) \quad (11)$$

The second step is the creation of the target pattern instances. Matched transformation rules, execute their action parts creating all target pattern instances and their model elements. Every pattern is characterized by the mandatory model elements that define the essence of the pattern, or what makes this pattern different from other patterns. Redefining (8) for use in (9) results with

$$mr_i(p_{So}) \leftarrow a_{i,1}(y_{i,1}, p_{So}) \wedge \dots \wedge a_{i,m(i)}(y_{i,m(i)}, p_{So}) \quad (12)$$

where $m(i)$ is a number of atoms in i-th mandatory transformation rule. Mandatory model elements in the supplied source pattern instances for the applied transformation tr, can be expressed as

$$me(p_{So}, tr) = \bigcup_{i=1}^{|mtr\,(tr)|} \bigcup_{j=1}^{m\,(i)} y_{i,j} \quad (13)$$

Whether a model element in the set of target patterns created by the applied transformation is mandatory or not, can be determined only in the context of another transformation from the library. However, in the context of the transformation that created the set of target pattern instances, all model elements created by mandatory transformation rules of the applied transformation are considered for mandatory model elements.

The last step is to create a set of constraints that will disallow designers to change some of the model elements in the involved pattern instances. Transformation binds involved pattern instances together by imposing constraints on their model elements. Each pattern instance can be bound with other pattern instances through several different transformations. Constraints are imposed by the mandatory transformation rules.

Imposed constraints are used to limit designer changes in the modeling space to prevent:

1. Violating correctness and completeness of the pattern instances by changing their mandatory model elements. Obviously, all mandatory model elements must be constrained.
2. Breaking transformation bindings by changing model elements that match source and target side of the mandatory transformation rules. In this case, constrained model elements do not need to be mandatory.

One constraint can be applied to a set of model elements. Each constraint also must contain a set of forbidden actions. At the moment, it is expected that constraining updating and deleting specific model elements is sufficient.

Let us define an involved pattern instance made of l model elements

$$p_i = \{e_1, e_2, \dots, e_l\} \subseteq \bigcup M_S \quad (14)$$

and a finite set of transformations applied to p_i

$$tr_{applied}\,(p_i) = \{tr_1(\mathrm{p_i}), tr_2(\mathrm{p_i}), \dots, tr_k(\mathrm{p_i})\} \quad (15)$$

From (14) and (15), we can derive a mapping function

$$C: tr_{applied}\,(p_i) \rightarrow X \quad (16)$$

where $X \subseteq p_i$ is a set of model elements in p_i constrained by all transformations from (15). Every pair of applied transformations can constrain a different subset of model elements in p_i

$$C(tr_j(\mathrm{p_i})) \cap C(tr_k(\mathrm{p_i})) = \emptyset \wedge j \neq k \quad (17)$$

In the context of (13) and (15), a set of mandatory model elements of the pattern instance p_i can be defined as

$$me(p_i) = me(p_i, tr_{applied}\,(p_i)) = \bigcup_{\mathrm{j}=1}^{\mathrm{k}} me(p_i, tr_j(p_i)) \quad (18)$$

having the following condition satisfied

$$C(tr_j(p_i)) = me(p_i) \wedge C(tr_k(p_i)) \cap me(p_i) = \emptyset \quad (19)$$

The conclusion is that the set of mandatory model elements for (14) is just a subset of constrained model elements by the set of applied transformations.

$$me(p_i) \subseteq \bigcup_{j=1}^{k} C(tr_j) \quad (20)$$

Each pattern instance can be a result of several different pattern instances done earlier in the same project, or it can be a reason for creating several new pattern instances later in the same project. Several good examples can be found in [14]: a facade associated with a web service client can be used as a mediator between two different subsystems. In this example, the mediator is the pattern whose instance is bound by two different transformations.

Definition: The measure of transformation applicability
The measure of transformation applicability is a percentage of transformation's mandatory rules that match supplied source pattern instances.

We already defined transformation applicability in (11). If not all of the mandatory transformation rules are matching supplied source pattern instances, then the transformation cannot be applied. The information on how much and which rules are not matched can be very valuable for a designer. This way, the designer can see how to elaborate piece of the information system design he is working on, in order to proceed in the pattern sequence. If we define a subset of mandatory transformation rules that are matching supplied source pattern instances as

$$mtr_{matc\,hed}(tr) \subseteq mtr(tr) \quad (21)$$

then the measure of transformation applicability can be expressed as

$$MA(mtr(tr), p_{So}) = \frac{|\mathrm{mtr}_{\mathrm{matched}}(\mathrm{tr})|}{|\mathrm{mtr}(\mathrm{tr})|} \quad (22)$$

Of course, the rest of the set of mandatory transformation rules, i.e., those that are not matched

$$mtr(tr) \setminus mtr_{matc\,hed}(tr) \quad (23)$$

can be used to determine what exactly is missing in the information system design. Consulting the measure of transformation applicability is one aspect of the design guidance.

The applicability of a pattern is previously considered by Fehling et al. in [13]. The pattern library presented in their article can give the applicability of a pattern based on the context which needs to be supplied manually. In the proposed method, the context is calculated from the model space, i.e., the model space makes a transformation in the modeling library applicable or not.

2) *Pattern instance elaboration*

A transformation can be used to perform changes on involved pattern instances. This approach is used when new pattern instances are created, or existing instances are updated or deleted. Even when two pattern instances are bound with the transformation, the source pattern instance can be elaborated by adding new details and model elements. A transformation can be made so that these newly added details automatically update the target pattern instance.

Model elements that are not constrained by one of the binding transformations are handled by optional transformation rules responsible for spreading of elaboration details. Bidirectionality is a very important transformation aspect described in [27] and [28]. While transformation might constrain changes of some model elements in target pattern instances, changes of unconstrained model elements in pattern instances across the modeling space are encouraged. Such changes must be propagated throughout the modeling space, wherever transformation between pattern instances allows it. This propagation must be automatic and seamless.

3) *Top-level pattern instances*

Top-level pattern instances do not have predecessors. These pattern instances can be modeled manually by a designer without using any transformation, instantiated directly from the modeling library, or they can be created by using a transformation.

If a top-level pattern instance is instantiated directly from the modeling library, then all model elements from the selected pattern are copied from the MOF repository directly to the model in the model space. Of course, the instantiation process must rename the selected pattern model elements, and impose constraints on the newly created pattern instance. In order to know which model elements are constrained, this information must be kept together with the solution of a pattern in the MOF repository of the modeling library.

If a transformation is used, such transformation does not need to have input source pattern instances. In order to give the transformation some instructions, input parameters can be used. Transformations that create only target pattern instances can be used both for validation and enforcement purposes. All transformation rules in this transformation are mandatory transformation rules that create an initial version of target pattern instances, and impose constraints on them. Obviously, these mandatory transformation rules are always matched, even when there is no supplied set of source pattern instances. However, imposed constraints must allow elaboration of newly created top-level pattern instances in order to allow adding needed details. Functional and non-functional requirements are typical examples of top-level patterns. An external service definition is another example of such pattern.

V. TRANSFORMATION AND TRACING LANGUAGE

Relationship between model elements and a pattern instance is not established within the UML. Although there is the *Package* element defined within the UML, its purpose is not the same as "the pattern instance". Also, transformation application and imposing constraints on involved pattern instances must leave some trail. Creation of a Transformation and Tracing Model (TTM), either

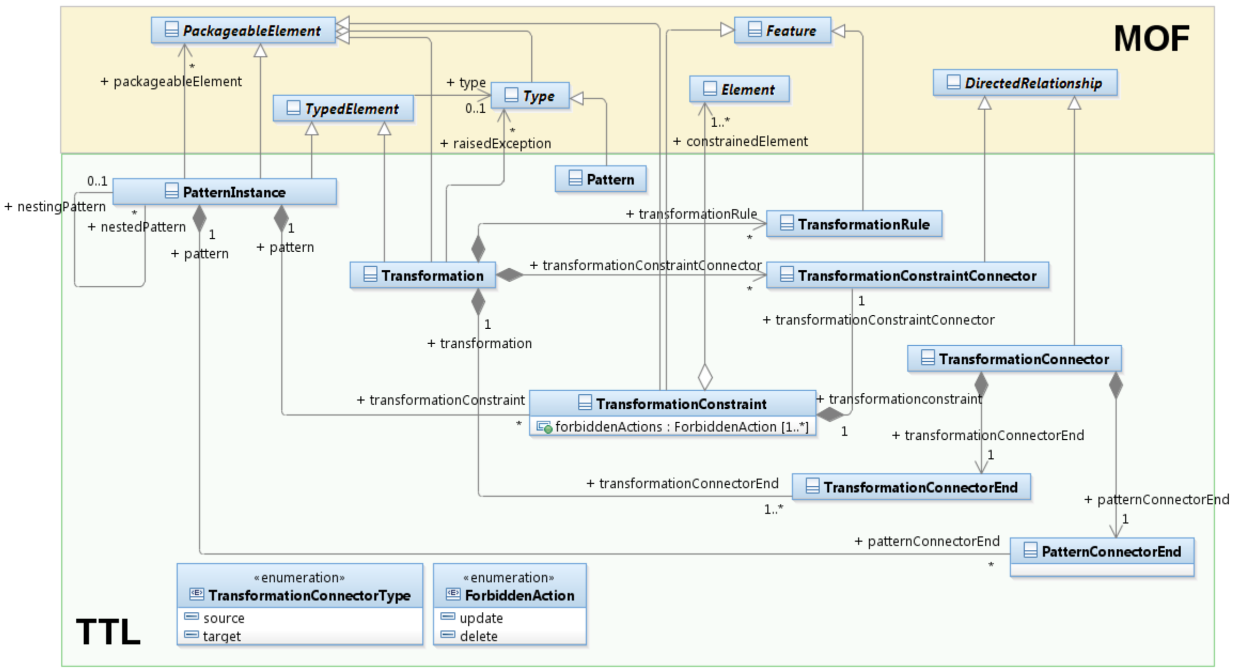
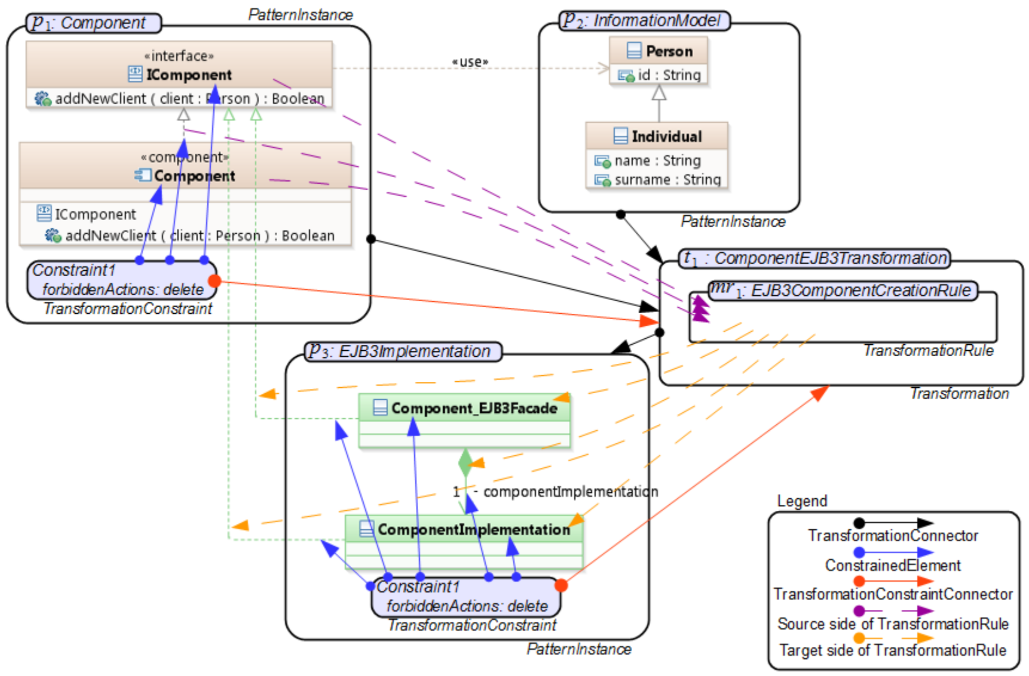


Figure 4. The Transformation and Tracing Language

automatically or manually, can help to resolve before mentioned issues. Every time a new pattern instance is created, a new model element is added into TTM representing this pattern instance. All model elements belonging to this pattern instance are automatically bound to it. It can be the result of the transformation, or it can be done manually. In both cases, the modeling tool must have capabilities for it. Also, each time when a transformation is used, this transformation is added to TTM including all relationships between pattern instances and used transformation. Each time a transformation is used, and this transformation is imposing constraints on involved pattern instances, these constraints are added to pattern instances in TTM and related to the transformation that created them. In order to do this kind of model, a Transformation and Tracing Language (TTL) must be defined. The UML and the TTL must be compatible, meaning that they must have a common M0 ancestor [28]. Therefore, the TTL must be a MOF metamodel. An overview of the TTL is presented in Figure 4.

The TTL is having the following elements:

1. *Pattern* - A pattern type. Allows classification of pattern instances.
2. *PatternInstance* - An element similar to the UML *Package* element. Represents a container for model elements. This element is defined by its name and type. Pattern type (or class) can be very helpful when constructing transformation rules, and it can impact the transformation applicability since transformations can be applied to the pattern instances of specific types.
3. *Transformation* - An element defined by its name and type, representing applied transformation, defined in (6) and (7). It contains transformation rules used in the transformation, here represented by the element *TransformationRule*. The transformation must be connected to a set of source and target pattern instances, being connected to at least one target pattern instance. Connector direction is determined by the *TransformationConnectorType* enumeration.
4. *TransformationConnector*, *TransformationConnectorEnd*, *PatternConnectorEnd* - A connector is a directed relationship between a pattern instance and a transformation. Connector direction must have a visual notation. If the connector is directed from the pattern instance to the transformation, it represents the source pattern instance in the context of the transformation. If the connector is directed from the transformation to the pattern instance, it represents the target pattern instance in the context of the transformation. Connector end elements represent the point of touch between the connector and the pattern instance, or the connector and the transformation.
5. *TransformationConstraint* - An element defined by its name, representing a constraint on members of a pattern instance imposed by used transformation. At least one model element in the pattern instance needs to be constrained. Also, a transformation must contain a set of forbidden actions, i.e., actions on constrained model elements that must be prevented by a tool. This element is contained by the pattern



p_1: Component
PatternInstance
«interface» IComponent
addNewClient (client : Person) : Boolean
«component» Component
IComponent
addNewClient (client : Person) : Boolean
Constraint1 forbiddenActions: delete
TransformationConstraint
«use»
p_2: InformationModel
Person
id : String
Individual
name : String
surname : String
PatternInstance
t_1 : ComponentEJB3Transformation
tr_1: EJB3ComponentCreationRule
TransformationRule
Transformation
p_3: EJB3Implementation
Component_EJB3Facade
1 - componentImplementation
ComponentImplementation
Constraint1 forbiddenActions: delete
TransformationConstraint
PatternInstance
Legend
TransformationConnector
ConstrainedElement
TransformationConstraintConnector
Source side of TransformationRule
Target side of TransformationRule

Figure 5. Example of a Transformation and Tracing Model

instance, and connected to the transformation responsible for the creation of the constraint. This element is the result of the transformation, and can be used to validate the pattern instance correctness and completeness.

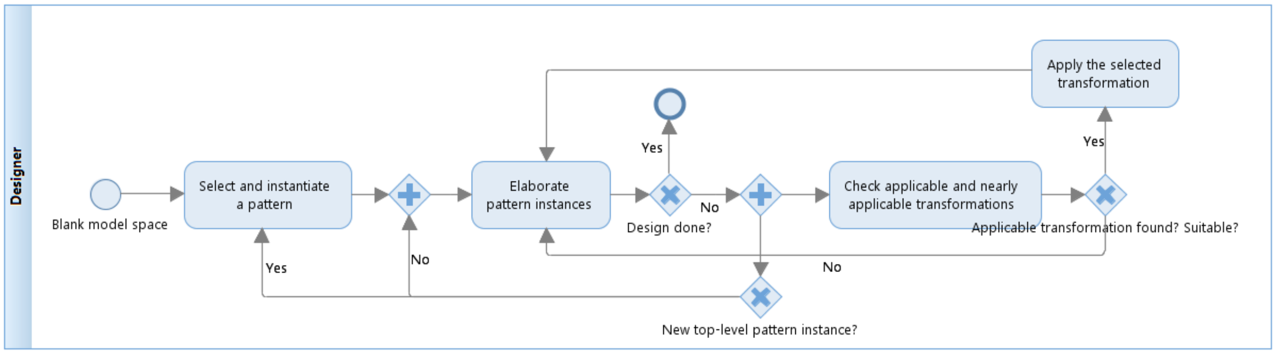
6. *TransformationConstraintConnector* - A relationship between resulting constraint and the transformation that created it, directed from the transformation to the constraint. Each constraint can be imposed by only one transformation, but one transformation can impose multiple constraints within multiple pattern instances.

In the TTM example in Figure 5, model elements in pattern instances p_1 and p_2 were created before t_1 was applied. We can say that pattern instances p_1 and p_2 were designed manually. Model elements in the pattern instance p_3 are produced by the transformation t_1. Actions taken during an information system design are automatically stored to a TTM for multiple purposes: preserving correctness and completeness of the modeling space, reconstruction of activities in the design process, and analysis of the resulting design work.

VI. Design Practice

The definition of the term "design practice" is given in [2]. A common situation is having to explain to designers what is the preferred design practice, and how an information system design should look like? The answer to this question is also the answer to the design approach classification framework given in [2].

Many companies have well established design practices, from the methodology, project activities, and modeling point of view. The selection of architectures, technology, and practical experience gives a company starting point in the information system design. The idea is to take this experience, put it into the modeling library in the form of



Designer
Blank model space
Select and instantiate a pattern
Elaborate pattern instances
Design done?
Yes
No
Check applicable and nearly applicable transformations
Applicable transformation found? Suitable?
Yes
No
Apply the selected transformation
New top-level pattern instance?
Yes
No

Figure 6. The proposed method

patterns and transformations, i.e., create a design knowledge system.

The way of applying this knowledge is very important as well. The design practice method proposed in this article, and outlined in Figure 6, consists of the following elements:

- Task 1: Selection and instantiation of appropriate pattern from the modeling library.
- Task 2: Elaboration of pattern instances.
- Task 3: Checking transformations from the modeling library that are applicable or nearly applicable onto elaborated pattern instances. If such transformation cannot be found, return to elaboration in task 2.
- Task 4: Transformation of the pattern instance by selecting applicable transformation from the modeling library. Continue on task 2 with newly created pattern instances.

Using the process in Figure 6 will create a pattern sequence. Depending on patterns and transformations in the modeling library, a big set of potential pattern sequences can be generated. Giving guidance to designers on a project means selecting appropriate pattern sequences from the set of all potential pattern sequences. By selecting appropriate pattern sequences, the design process is directed into the desired direction and outcome.

A. Guidance given through the modeling library

The modeling library is comprised of patterns and transformations. Since a transformation binds two pattern instances together (as described in Section III), selection of a transformation imposes a selection of involved patterns. Similarly, a selection of patterns imposes a selection of potentially applicable transformations.

Applicability and the measure of transformation applicability are important transformation features that can be used to form a pattern sequence. A designer can elaborate a model or a pattern instance, and occasionally check for transformations that are applicable to the model or pattern instance he is working on. If there is no transformation currently applicable, the designer can check transformations that are nearly applicable, and the gap that needs to be closed in the model or the pattern instance in order for this nearly applicable transformation to become applicable. Of course, many designers have enough experience to know which transformation would need to be used next, even before modeling of the pattern instance is finished. If there is a problem with selected transformation, and transformation rules in the transformation are not correct, meaning that the transformation will never become applicable, this particular transformation can be changed as part of the design practice evolution.

Giving guidance means selecting transformations from the modeling library that will be used in the project. A design lead can manage the set of allowed transformations for the project, limiting designer's choice of applicable or nearly applicable transformations. For example, the architectural decision will influence the choice of transformations for the project. Similarly, the design lead can manage a set of allowed patterns that are going to form the pattern sequences in the project, and by doing that implicitly to select a set of allowed transformations.

B. Guidance given through a model

More specific guidance can be given through a specific model that predetermines patterns and transformations used in the information system design process. Such model is created *a priori*, before the start of the design activities. Creation of the guidance model is an ongoing activity through the whole project. The TTL can be used for this purpose. This model must represent a selection of allowed pattern types and related transformations. Such model can be used by a designer to check guidance, or directly by a modeling tool for selection of allowed transformation list for particular pattern type. It is the same approach as in the previous section, with additional visualization of selected design practice for the project.

VII. EXAMPLE: BUSINESS PROCESS ORIENTED SYSTEM

The example in Figure 3 is business process oriented. The common name for this kind of system is Business Process Management (BPM) System. In this section, a detailed walk through for the example in Figure 3 is given. The first step is to create a business process model. Such business process model can be done using BPMN [24].

In this case, a very simple business process is modeled from scratch. A new pattern instance p_1 is created and placed in the TTM. When modeling, the model elements of the business process are associated with the pattern instance p_1. The business process contains one human task having the

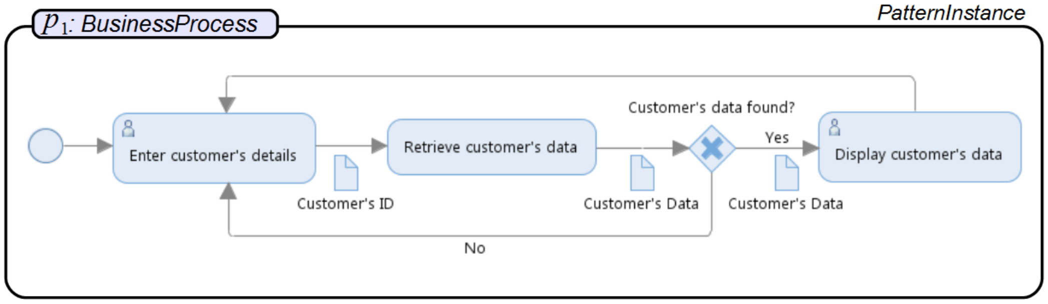


Figure 7. The business process

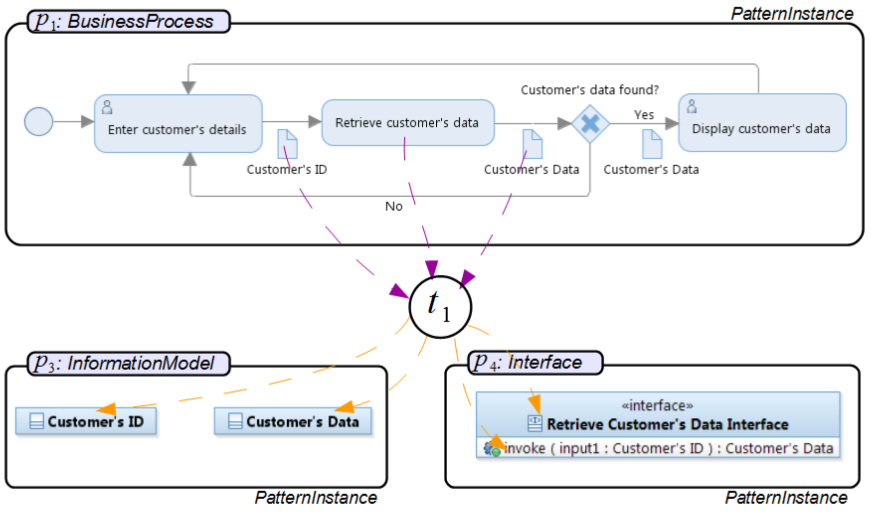


Figure 8. Transformation of the task to entities and the interface

user interface, where a user must enter customer's identification data, such as "social id", "account number", or something else. The customer's identification is then sent to the automatic task named "Retrieve customer's data", which invokes an information service that finds and read the data. If the identified customer cannot be found in the system, a null is returned back from the invoked service. Returned data are then tested, and if customer's data exists, it is displayed in the human task "Display customer's data".

Each automatic task contains a signature that involves input parameters and output results. These details must be observed on a correctly modeled business process, as in Figure 7. This signature is used for transforming this automatic task into a number of pattern instances, containing entities and interfaces needed for building the service that will be invoked by this task. Each business process can have more than one automatic task. This means that one transformation from the modeling library can be applied more than once per one pattern instance. A designer must be presented with the list of applicable transformations along with all details, including model elements in source pattern instances that can be used in the transformation. In case of a business process that contains more than one automatic task, an applicable transformation can be applied to each automatic task.

Figure 8 presents a transformation from the automatic task in the business process into a set of entities and an interface. After applying the transformation t_1, the pattern instance p_3 is created. Along with the pattern instance, model elements representing entities of two business objects constituting information flow of the transformed task are created. The transformation t_1 must create constraints that will prevent deleting business objects, the task in the business process, and both of the entities in the pattern instance p_3.

The transformation t_1 is also responsible for creation of the pattern instance p_4, which consists of model elements that represents the interface of the task in the business process: one operation receiving the input business object as the input parameter, and returning the output business object as the result. The transformation t_1 must create constraints that will prevent deleting the involved model elements in pattern instances p_3 and p_4. Another constraint that will prevent direct updating the operation on the interface must be created by the transformation t_1 as well, because updates on transformation's target pattern instances must be result of the elaboration of the source pattern instances. It is worth noticing that an operation and comprising parameters are, according to the MOF, not the same model elements. While an operation can be constrained, comprised parameters can be updated by the transformation if there are changes on business objects in the business process.

The next step is to elaborate the pattern instance p_3. As presented in Figure 9, a designer is filling additional details for entities in the pattern instance p_3. These details include attributes and their types for each entity.

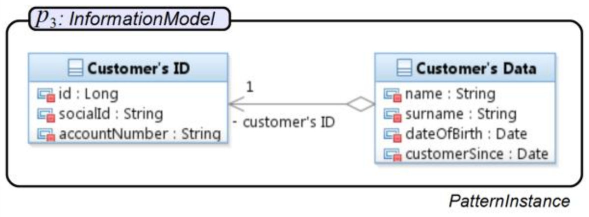


Figure 9. Elaborated entities

At the same time, a lead IT architect is defining the architecture of the information system. It is very important that the architecture is a part of the modeling space, so that it can be used by the proposed method for selection of allowed transformations. This way, the architecture guides the design process as well.

The architecture presented in Figure 10 defines a couple of very important elements for the selection of applicable transformations. First, JAX-WS2 will be used for invoking the service from the automated task in the business process. Second, the service will be deployed on Java Enterprise Edition application server. Third, entities will become tables in the DB2 database.

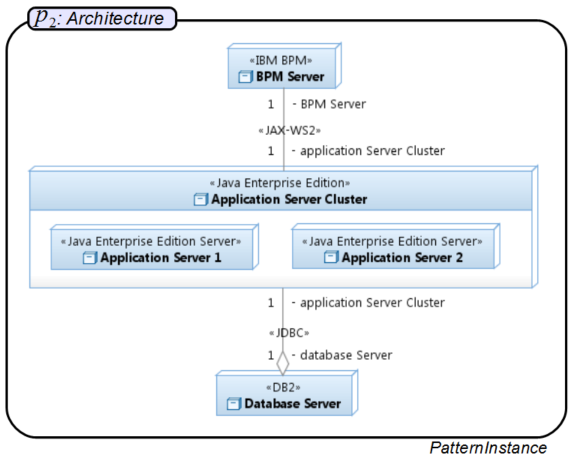

p_2: Architecture
«IBM BPM»
BPM Server
1 - BPM Server
«JAX-WS2»
1 - application Server Cluster
«Java Enterprise Edition»
Application Server Cluster
«Java Enterprise Edition Server»
Application Server 1
«Java Enterprise Edition Server»
Application Server 2
1 - application Server Cluster
«JDBC»
1 - database Server
«DB2»
Database Server
PatternInstance

Figure 10. The architecture

Figure 10 is one pattern instance. This would suggest that the whole architecture is placed in the single pattern instance container. In fact, Figure 10 represents a typical business process oriented architectural pattern specified for the concrete environment and products.

So far, the design is still not platform specific. Pattern instances p_3 and p_4 can be transformed into a component.

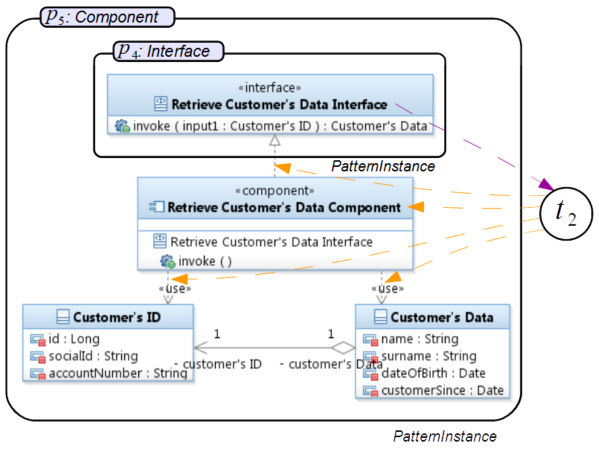


Figure 11. The component

Such transformation must create a component, and define that it is realized using already created interface, as in Figure 11. For reference purpose, transformation t_2 adds dependencies between the created component and entities contained in the realized interface.

After this, work on the platform specific design can begin. The next step is to elaborate additional details in the pattern instance p_3 in order to make transformation t_3 applicable.

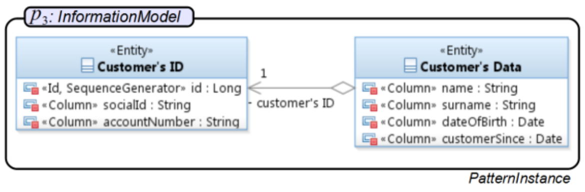
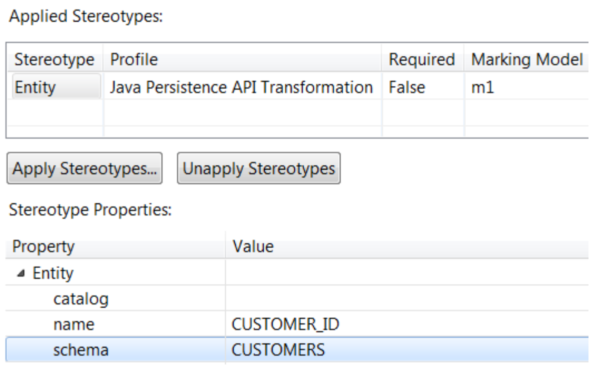

p_3: InformationModel
«Entity»
Customer's ID
«Id, SequenceGenerator» id : Long
«Column» socialId : String
«Column» accountNumber : String
1
- customer's ID
«Entity»
Customer's Data
«Column» name : String
«Column» surname : String
«Column» dateOfBirth : Date
«Column» customerSince : Date
PatternInstance

Figure 12. Entities with JPA stereotypes

During elaboration, entities in the pattern instance p_3 are enriched with JPA related stereotypes, as in Figure 12. Since transformation t_3 takes in consideration model elements marked with *Entity* stereotype, this elaboration is needed in order to make the transformation applicable. During the elaboration of entities, a designer must also define additional properties for applied stereotypes. For example, a *Column* stereotype has a set of very important properties that need to be defined, and can be used by transformations that will be applied next. Figure 13 presents a set of properties for the *Entity* stereotype.


Applied Stereotypes:

Stereotype	Profile	Required	Marking Model
Entity	Java Persistence API Transformation	False	m1

Stereotype Properties:

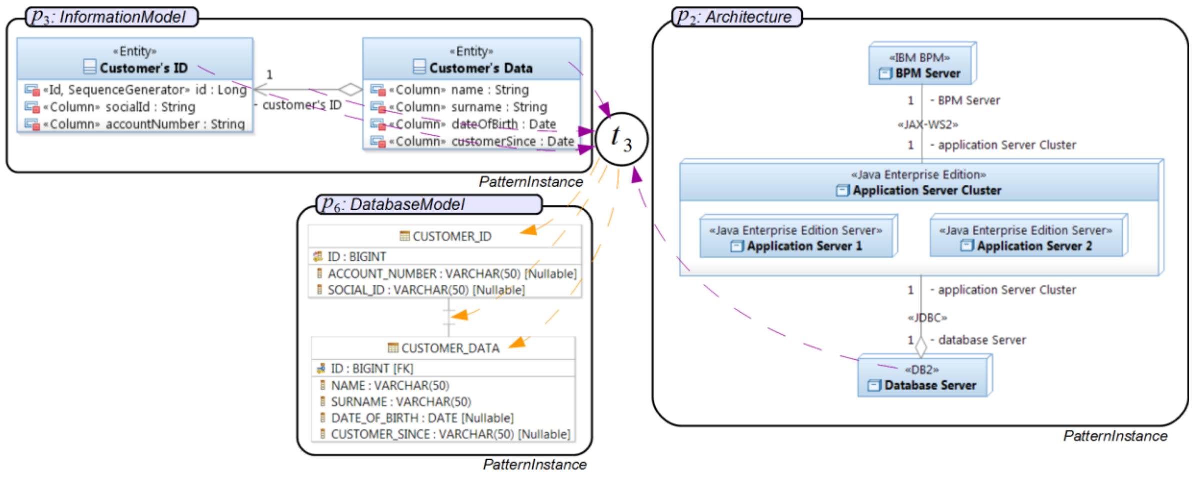
Property	Value
◢ Entity	
catalog	
name	CUSTOMER_ID
schema	CUSTOMERS

Figure 13. Entity stereotype properties

However, this elaboration makes the pattern instance p_3 more platform specific than platform independent. Although there is still no precise definition about the concrete database that will be used, this information can be found in the architecture contained in the pattern instance p_2.

Figure 14 is the result of applying transformation t_3 to the pattern instances p_2 and p_3. The transformation takes only entities marked with *Entity* stereotypes. All properties of applied stereotypes are used in the transformation. Also, the transformation matches the database node in the architecture pattern instance p_2 . The result of the transformation is the pattern instance p_6 that comprises a number of DB2 specific model elements, representing database tables of entities.

In case of having an Oracle database node in the architecture, another transformation would become applicable, which would create Oracle specific model elements in the pattern instance p_6.



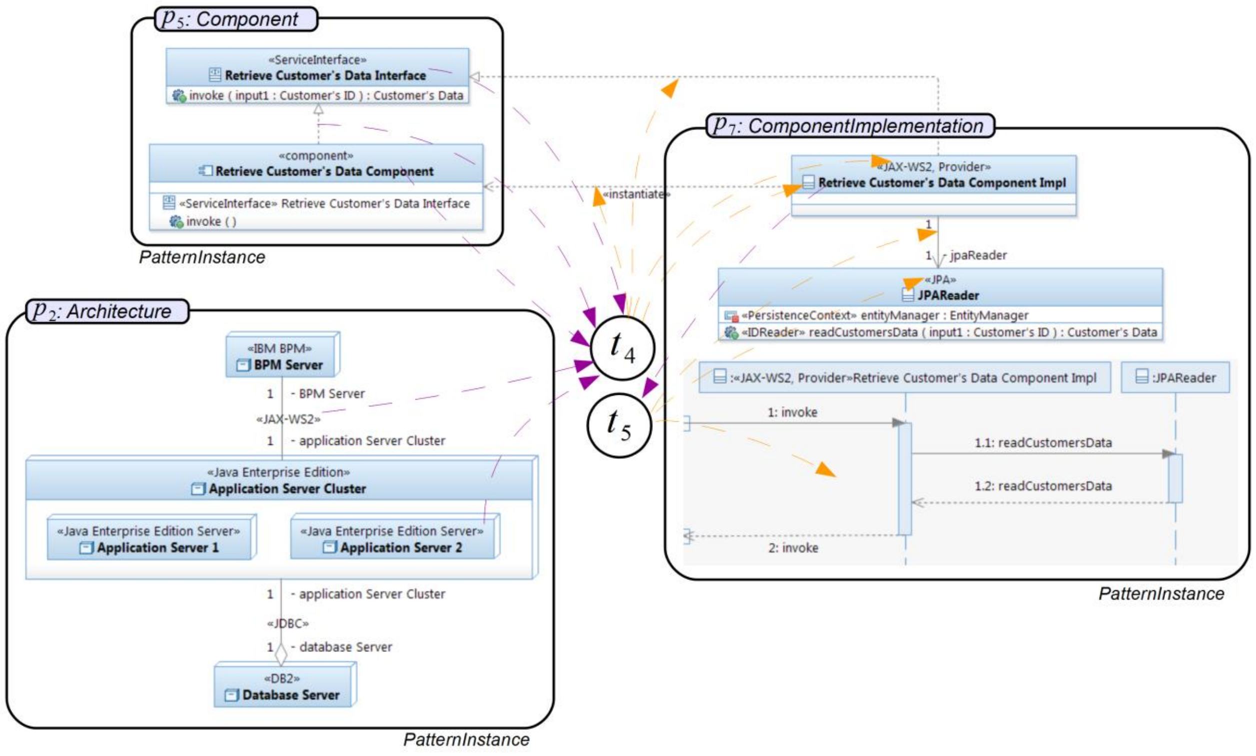
p_3: InformationModel
«Entity» Customer's ID
«Id, SequenceGenerator» id : Long
«Column» socialId : String
«Column» accountNumber : String
«Entity» Customer's Data
«Column» name : String
«Column» surname : String
«Column» dateOfBirth : Date
«Column» customerSince : Date
1
- customer's ID
t_3
PatternInstance
p_6: DatabaseModel
CUSTOMER_ID
ID : BIGINT
ACCOUNT_NUMBER : VARCHAR(50) [Nullable]
SOCIAL_ID : VARCHAR(50) [Nullable]
CUSTOMER_DATA
ID : BIGINT [FK]
NAME : VARCHAR(50)
SURNAME : VARCHAR(50)
DATE_OF_BIRTH : DATE [Nullable]
CUSTOMER_SINCE : VARCHAR(50) [Nullable]
PatternInstance
p_2: Architecture
«IBM BPM» BPM Server
1 - BPM Server
«JAX-WS2»
1 - application Server Cluster
«Java Enterprise Edition» Application Server Cluster
«Java Enterprise Edition Server» Application Server 1
«Java Enterprise Edition Server» Application Server 2
1 - application Server Cluster
«JDBC»
1 - database Server
«DB2» Database Server
PatternInstance

Figure 14. Entities with JPA stereotypes

The pattern instance p_6 is a true example of a platform specific design. Now that the information part of the design is done, designing component details is the next step.

The component implementation presented in Figure 15 is a result of two steps. The first step is applying transformation t_4 for creation of a component implementation. This transformation is applicable only when *JAX-WS2* and *Java Enterprise Edition Server* stereotypes are found in the architecture pattern instance. It creates a class with appropriate stereotypes for further transformations, an interface realization between the newly created class and the component's interface, and a relationship that marks the newly created class as an instance of the component. However, transformation t_4 did not create any specific implementation details. Everything that was created is the class will be eventually transformed into a JAX-WS2 web service provider.

The second step is applying transformation t_5 that adds implementation details to the service provider created in the previous step. Again, a designer can have a number of transformations at disposal that can look for certain model elements in the existing pattern instances. In this case, transformation t_5 takes the signature of the *invoke* method, parameter types and creates a method in a new JPA reading



p_5: Component
«ServiceInterface» Retrieve Customer's Data Interface
invoke (input1 : Customer's ID) : Customer's Data
«component» Retrieve Customer's Data Component
«ServiceInterface» Retrieve Customer's Data Interface
invoke ( )
PatternInstance
«instantiate»
p_7: ComponentImplementation
«JAX-WS2, Provider» Retrieve Customer's Data Component Impl
1
1 - jpaReader
«JPA» JPAReader
«PersistenceContext» entityManager : EntityManager
«IDReader» readCustomersData (input1 : Customer's ID) : Customer's Data
:«JAX-WS2, Provider»Retrieve Customer's Data Component Impl
:JPAReader
1: invoke
1.1: readCustomersData
1.2: readCustomersData
2: invoke
PatternInstance
t_4
t_5
p_2: Architecture
«IBM BPM» BPM Server
1 - BPM Server
«JAX-WS2»
1 - application Server Cluster
«Java Enterprise Edition» Application Server Cluster
«Java Enterprise Edition Server» Application Server 1
«Java Enterprise Edition Server» Application Server 2
1 - application Server Cluster
«JDBC»
1 - database Server
«DB2» Database Server
PatternInstance

Figure 15. The component implementation

helper class. This method is marked with the *IDReader* stereotype, which can be used later to transform this method into a code snippet. Transformation t_5 also creates an association between the component's implementation and the newly created JPA reading helper class. A collaboration sequence between the component's implementation class and the JPA reading helper class is created as well. This collaboration sequence can be used in creating the code later on.

As mentioned in [26], transformations from model to code need to be treated slightly differently. Template based approach is suitable for this example.

Source Code 1: Transformed JPA entity

```
@Table(name = "CUSTOMER_ID", schema = "CUSTOMERS")
@Entity
public class CustomersID implements Serializable {

   private static final long serialVersionUID = 0;

   public CustomersID() {}

   @Id
   private Long id;

   @Column(nullable = false, columnDefinition =
         "SOCIAL_ID", length = 50)
   private String socialId;

   @Column(nullable = false, columnDefinition =
         "ACCOUNT_NUMBER", length = 50)
   private String accountNumber;

   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }

   public String getSocialId() {
      return socialId;
   }

   public void setSocialId(String socialId) {
      this.socialId = socialId;
   }

   public String getAccountNumber() {
      return accountNumber;
   }

   public void setAccountNumber(String accountNumber) {
      this.accountNumber = accountNumber;
   }
}
```

The listing in Source Code 1 is the final result of transforming from entities in the pattern instance p_3. Such transformations can be applicable only on *InformationModel* pattern instances, i.e., including only model elements that are contained in specific pattern instances.

Using pattern instance p_6 and the applicable transformation, the following SQL script is generated.

SQL Script 1: Database DDL script

```
CREATE SCHEMA "CUSTOMER";

CREATE TABLE "CUSTOMER"."CUSTOMER_DATA" (
            "NAME" VARCHAR(50) NOT NULL,
            "SURNAME" VARCHAR(50) NOT NULL,
            "DATE_OF_BIRTH" DATE,
            "CUSTOMER_SINCE" VARCHAR(50),
            "ID" BIGINT NOT NULL
        )
        DATA CAPTURE NONE;

CREATE TABLE "CUSTOMER"."CUSTOMER_ID" (
            "ID" BIGINT NOT NULL GENERATED BY
DEFAULT AS IDENTITY ( START WITH 1 INCREMENT BY 1
MINVALUE 1 MAXVALUE 9223372036854775807 NO CYCLE CACHE
20),
            "ACCOUNT_NUMBER" VARCHAR(50),
            "SOCIAL_ID" VARCHAR(50)
        )
        DATA CAPTURE NONE;

ALTER TABLE "CUSTOMER"."CUSTOMER_DATA" ADD CONSTRAINT
"CUSTOMER_DATA_PK" PRIMARY KEY
        ("ID");

ALTER TABLE "CUSTOMER"."CUSTOMER_ID" ADD CONSTRAINT
"CUSTOMERS_ID_PK" PRIMARY KEY
        ("ID");

ALTER TABLE "CUSTOMER"."CUSTOMER_DATA" ADD CONSTRAINT
"CUSTOMER_DATA_CUSTOMER_ID_FK" FOREIGN KEY
        ("ID")
        REFERENCES "CUSTOMER"."CUSTOMER_ID"
        ("ID")
        ON DELETE CASCADE;
```

Finally, pattern instances p_5 and p_7 can be transformed into the web service that can be called from the task in the business process.

Source Code 2: The web service interface

```
@WebService(targetNamespace="customer")
public interface RetrieveCustomersDataInterface {
   @WebMethod
   public CustomersData invoke(CustomersID input1);
}
```

Source Code 3: The web service

```
@WebService(targetNamespace="customer")
public class RetrieveCustomersDataComponentImpl
implements RetrieveCustomersDataInterface {
   private JPAReader jpaReader;

   public CustomersData invoke(CustomersID input1) {
      // begin-user-code
      return jpaReader.readCustomersData(input1);
      // end-user-code
   }
}
```

Source Code 4: The JPA reader

```
public class JPAReader {
   @PersistenceContext
   private EntityManager entityManager;
   // TODO Finish instancing entity manager

   public CustomersData readCustomersData(CustomersID
         input1) {
      // begin-user-code
      Query q=null;
      if(input1!=null && input1.getId()!=null) {
         q=entityManager.createQuery("select obj from
               CustomersData obj where
               obj.customersID.id = :id");
         q.setParameter("id", input1.getId());
      } else if(input1!=null &&
         input1.getSocialId()!=null) {
         q=entityManager.createQuery("select obj from
               CustomersData obj where
               obj.customersID.socialId = :socialId");
         q.setParameter("socialId",
               input1.getSocialId());
```

```
            } else if(input1!=null &&
               input1.getAccountNumber()!=null) {
                q=entityManager.createQuery("select obj from
                        CustomersData obj where
                        obj.customersID.accountNumber =
                        :accountNumber");
                q.setParameter("accountNumber",
                        input1.getAccountNumber());
            }
            if(q!=null) {
                return (CustomersData)q.getSingleResult();
            }
            return null;
            // end-user-code
        }
    }
```

The call between the component's implementation and the JPA helper class is transformed from the collaboration sequence. All additional details in the code, such as annotations, are added from stereotype information. Stereotypes on the JPA helper class help to select an appropriate template for the code.

VIII. Conclusion and Future Work

We have demonstrated that even such small example can be full of details and rules, enforcing us to use specific model elements, stereotypes, and patterns. It is obvious that patterns are not just a couple of documented ways of solving problems, which can be found in the books. It is everything that we want to use to repeat our solutions. Patterns are a good start for defining our designing practice.

MDA has two major practical problems: designers have too much freedom while creating the information system design so that the transformation scope can become very ambiguous. Usage of a pattern as the main building element for the information system design is a well known approach. In the context of this article, design of an information system is done block by block by reusing patterns, allowing a design lead to choose blocks to be used. Such approach allows a design team to use past positive experience to select or define best patterns for the information system they are designing. This approach also helps to build pattern sequences that can fit into a design and development methodology used for the project.

The novelty introduced in this article is the way of building pattern sequences through use of transformation, an approach typically used in the MDA. Applicability and the measure of applicability are very important features of the transformation definition, given in this article. They enable controlled application of transformations, which represents guidance for the design team. They also represent a way how new designers can learn the established design practice.

Of course, designers are still free to model according to their preferences, as long as they are within boundaries imposed by the proposed method, which is assured by an optional part of each transformation helping team to keep model elements of bound pattern instances synchronized. The bidirectionality feature of the transformation helps to reflect changes in both directions. Chains of pattern instances can be easily updated through transformations used to form a chain. Since a pattern instance is supposed to have smaller scope than a model, keeping several pattern instances synchronized during elaboration should be much easier than with big models.

The proposed method successfully answers challenges introduced in Section I. The modeling library contains design knowledge, and offers a selection of the design approach based on the current context in the modeling space. The article also successfully answers question of transformation reusability. The result of all these improvements is a higher quality of models comprising the design of an information system.

Current modeling tools are introducing a high level of automation. This automation is mostly related to elements of the modeling languages supported by a modeling tool. Changing the modeling tool behavior to follow the model in a modeling space is needed feature.

The TTL defined in this article can be extended with elements for interaction with modeling tool, model analysis capabilities, and model quality assessment. Interaction between a TTM and a modeling tool can be extended with modeling events, allowing a design lead to define modeling tool actions associated with patterns and transformations. For example, a TTM can include an event handler on a pattern that can be triggered by the modeling tool when a new subcomponent is added into a pattern instance. The event handler initiates execution of a specific transformation that automatically adds interface and interface realization relationship for this newly added subcomponent.

References

[1] D. Kreža and K. Fertalj, "A method for situational and guided information system design," Proceedings of the 6th International Conference on Pervasive Patterns and Applications, IARIA, May 2014, pp. 70-78.

[2] P. Ralph and Y. Wand, "A proposal for a formal definition of the design concept," in Design requirements engineering: A ten-year perspective, Springer Berlin Heidelberg, vol. 14, pp. 103-136, 2009, doi: 10.1007/978-3-540-92966-6_6.

[3] Object Management Group, "MDA guide, version 1.0.1, 2003". Available from http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf 2014.11.15

[4] M.F. Gholami and R. Ramsin, "Strategies for improving MDA-based development processes," Proceedings of the 2010 International Conference on Intelligent Systems, Modelling and Simulation (ISMS), IEEE, Jan. 2010, pp. 152-157, doi: 10.1109/ISMS.2010.38.

[5] L. Osterweil, "Software processes are software too, revisited: an invited talk on the most influential paper of ICSE 9," Proceedings of the 19th international conference on Software engineering ICSE '97, ACM, May 1997, pp. 540-548, doi: 10.1145/253228.253440.

[6] C. F. J. Lange and M. R. V. Chaudron, "Managing model quality in UML-based software development," 13th IEEE International Workshop on Software Technology and Engineering Practice, IEEE, Sep. 2005, pp. 7-16, doi: 10.1109/STEP.2005.16.

[7] F. Chitforoush, M. Yazdandoost, and R. Ramsin, "Methodology support for the model driven architecture," Proceedings of the 14th Asia-Pacific Software Engineering Conference, IEEE, Dec. 2007, pp. 454-461, doi: 10.1109/ASPEC.2007.58.

[8] I. Jacobson, G. Booch, and J. E. Rumbaugh, The unified software development process - the complete guide to the unified process from the original designers, Addison-Wesley, 1999.

[9] P. Kroll and P. Kruchten, The rational unified process made easy: a practitioner's guide to the RUP, Addison-Wesley, 2003.

[10] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

[11] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I.Fiksdahl-King, and S. Angel, A pattern language: towns, buildings, constructions, Oxford University Press, 1977.

[12] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck, "A collection of patterns for cloud types, cloud service models, and cloud-based application architectures". Available from http://www.cloudcomputingpatterns.org 2014.11.15

[13] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, "An architectural pattern language of cloud-based applications," Proceedings of the 18th Conference on Pattern Languages of Programs, ACM, Oct. 2011, pp. 2, doi: 10.1145/2578903.2579140

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software, 28th ed., Addison-Wesley, 2004.

[15] G. Hohpe and B. Woolf, Enterprise Integration Patterns, Addison Wesley, 2004.

[16] Object Management Group, "Meta object facility (MOF) core specification, version 2.4.2, 2014". Available from http://www.omg.org/spec/MOF/2.4.2/PDF/ 2014.11.15

[17] S.D. Frankel, Model driven architecture: applying MDA to enterprise computing, Wiley Publishing, 2003.

[18] M. Gupta, R. Singh Rao, and A. Kumar Tripathi, "Design pattern detection using inexact graph matching," 2010 International Conference on Communication and Computational Intelligence, IEEE, Dec. 2010, pp. 211-217.

[19] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," Proceedings of the 31st International Conference on Software Engineering, IEEE, May 2009, pp. 276-286, doi: 10.1109/ICSE.2009.5070528.

[20] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From pattern languages to solution implementations," Proceedings of the 6th International Conference on Pervasive Patterns and Applications, IARIA, May 2014, pp. 12-21.

[21] D.R. Stevenson, J.R. Abbott, J.M. Fischer, S.E. Schneider, B.K. Roberts, M.C. Andrews, D.J. Ruest, S.K. Gardner, and C.D. Maguire, "Pattern implementation technique," U.S. Patent 8 661 405, Feb. 25, 2014.

[22] R. Porter, J. O. Coplien, and T. Winn, "Sequences as a basis for pattern language composition," Science of Computer Programming, Elsevier, vol. 56, pp. 231–249, Apr. 2005.

[23] W. Hasselbring, "Component-based software engineering," Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing, vol. 2, pp. 289-305, 2002, doi: 10.1142/9789812389701_0013.

[24] Object Management Group, "Business process model notation, version 2.0.2, 2013". Available from http://www.omg.org/spec/BPMN/2.0.2/PDF/ 2014.11.15

[25] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. Varro, "Using graph transformation for practical model-driven software engineering," in Model-driven Software Development, Springer Berlin Heidelberg, pp. 91-117, 2005, doi: 10.1007/3-540-28554-7_5.

[26] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, vol. 45, no. 3, Oct. 2003, pp. 1-17.

[27] Object Management Group, "Meta object facility (MOF) 2.0 query/view/transformation (QVT), version 1.1, 2011". Available from http://www.omg.org/spec/QVT/1.1/PDF/ 2014.11.15

[28] A. G. Kleppe, J. B. Warmer, and W. Bast, MDA explained, the model driven architecture: Practice and promise, Addison-Wesley, 2003.

[29] F. Jouault and I. Kurtev, "Transforming models with ATL," Proceedings of the MoDELS 2005 Conference, Springer Berlin Heidelberg, Oct. 2005, pp. 128-138, doi: 10.1007/11663430_14.

[30] A. Van Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," Journal of the ACM (JACM), ACM, vol. 38, no. 3, pp. 619-649, Jul. 1991, doi: 10.1145/116825.116838.