A Method for Situational and Guided Information System Design

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*—Model Driven Architecture is not highly used in current information system development practice. One of the reasons is that modeling languages are mostly used for documenting of the information system development and enhancement of communication within project teams. Without guidance, an information system design results in models of low quality, which cannot be used for anything more than documenting pieces of the information system. When it comes to model transformation, project team members usually reuse predefined transformations included in a modeling tool. Models of low quality that are not traceable and not complete are hard to transform. Model quality cannot be high if modeling activities are not guided and constrained. Guidance and constraints can be imposed through project activities led by a senior designer responsible for model quality. In this article, a method for situational modeling guidance is presented. This method is adaptable and situation dependent. Implemented within a modeling tool, the method should allow project team members responsible for model quality to give guidance and constraints, and to ensure model quality through the modeling tool.**

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

I. INTRODUCTION

The Model Driven Architecture (MDA), standardized by the Object Management Group (OMG) [1], is an information system design approach based on models and model transformations. Using MDA, an information system is designed through several models of different abstraction levels, from business oriented models to technical and platform specific models. MDA defines three different types of models having different levels: abstract and business oriented Computational Independent Model (CIM), technically oriented Platform Independent Model (PIM), and very detailed Platform Specific Model (PSM).

Model transformation is a key procedure in MDA. According to the specification [1], "model transformation is the process of converting one model to another model". Model transformation can be done 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. Such an approach is defined within various design and development methodologies. Chitforoush, Yazdandoost and Ramsin [2] are giving an overview of MDA specific methodologies. Most of these methodologies were developed for specific projects. Some generic design and development methodologies, such as Rational Unified Process (RUP) [3][12], also rely on model based design. However, basic purpose of methodologies is to help organize projects, giving guidance for project activities and deliverables, leaving execution to project team members.

When a modeling language is structured and formal enough, automatic transformation can be used. The Meta Object Facility (MOF), standardized by the OMG and described in [5], is a metalanguage for modeling languages that can be transformed automatically. Automatic transformation takes artifacts of a source model and converts them into artifacts of a target model by using transformation mapping. Transformation can be additionally used to establish relationships between models, or to check consistency of artifacts between a source and target model. Czarnecki and Helsen [4] elaborate a number of model transformation approaches and basic features of transformation rules. 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) [7]. QVT Relational language (QVT-R) is a typical example of a declarative approach with a graphical notation. QVT Operational language (QVT-O) is an example of an imperative approach.

The focus of this article is delivery of the models that are of high quality. In order to understand what this means, we can use one of the existing quality models. One example is the quality model given by Lange and Chaudron [8]. Relying only on methodology guidance will not necessarily produce a model of high quality, because it allows designers to focus on wrong aspects and details within the model. The result can be a model of poor quality, problems with traceability and inability to transform or analyze created model. One way to solve these problems is by appointing a senior designer to the design lead role. The design lead responsibilities are to establish modeling guidance and constraints, oversee modeling work, check delivered models and to ensure model quality. According to the quality model [8], this means that all models are traceable, complete, consistent and correspondent to the information system. Establishing modeling constraints means imposing patterns that need to be used during the information system design.

 ISBN: 978-1-61208-343-8

In this article, a method for automated modeling guidance and imposing constraints is proposed. The proposed method utilizes existing specifications such as MDA, MOF, UML and QVT to achieve a guided process of an information system design, supported by model transformation. The proposed method will be extensible in order to naturally fit existing design and development methodologies. The purpose of the method is to ease communication between a design lead and his team members and to enable management of an information system design work through usage of a modeling tool.

In Section II, a modeling space is defined. The modeling space is a way how to combine all models of an information system together, giving them relationship and defining their purpose. In the same Section, a relationship between pattern instances and models is given. In Section III, current modeling practice in the context of methodologies is discussed, which helps understand how the pattern instances are created during 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 artifacts. In Section VI, an overview of the method for situational and guided information system design is given.

II. MODELING SPACE

A modeling space can be represented as a three dimensional space containing all possible models of a designed information system. The modeling space must follow MDA philosophy, support different levels of abstraction given in MDA specification, and classification of the containing models.

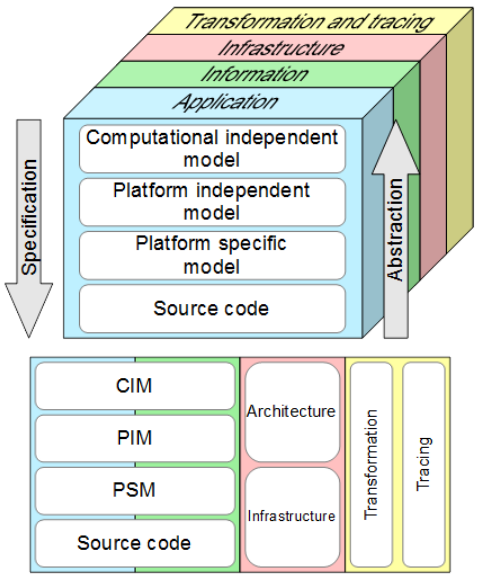

Figure 1. Structure of a modeling space

The proposed modeling space presented in Figure 1 is in three dimensions because it 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. And 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. The proposed 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. Figure 1 shows the placement of different MDA model types in the proposed modeling space.

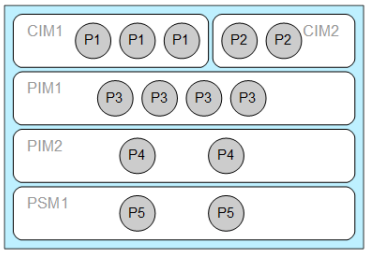

Figure 2. Models and pattern instances placed in the application layer of the modeling space

Each model is a set of artifacts. These artifacts originate from a modeling language, such as UML. A set of models together represent a design of an information system. However, there are building blocks between single artifact and a whole model that are meaningful for designers. These building blocks are patterns. Example of repeating patterns within different models is given in Figure 2. CIM1 contains repeating sets of model artifacts that can be interpreted as requirements, CIM2 contains business processes, PIM1 use cases, PIM2 components, and PSM1 implementation of components defined in PIM2.

Which patterns will appear in the modeling space depends on the design of an information system. Computational independent 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. Platform independent patterns are derived from architecture and computational independent patterns. They represent an elaboration of computational independent patterns within an architectural context. The most detailed are platform specific patterns that represent the implementation of platform independent patterns for a specific infrastructure yielded by the previously determined architecture.

In order to establish the method proposed in this article, a library of modeling patterns and transformations must be established. Modeling patterns can be determined in several different ways. Gamma, Helm, Johnson and Vlissides [9]

 ISBN: 978-1-61208-343-8

propose a list of basic object-oriented patterns visualized in the UML. Hohpe and Woolf [10] propose a list of enterprise integration patterns. Enterprise integration patterns are more abstract than object-oriented patterns.

Collecting modeling patterns from existing models of the already developed information system is another way. It can be done manually or automatically by detecting repetitions in existing models. Detection itself can be done by the graph matching method [11]. However, this is just a part of a collection process. Rahm et al. [15] propose 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.

And finally, as already mentioned, modeling patterns can be a great way how to give a sense of direction and cooperation to a team of designers. A pattern is a class, a blueprint that binds one or more modeling artifacts together. Application of a pattern means his instantiation within at least one model in the modeling space. Applying the modeling pattern does not mean that the modeling is completed. Adding details and further elaboration of the pattern instance is needed, in order to give it enough details to fit an information system design.

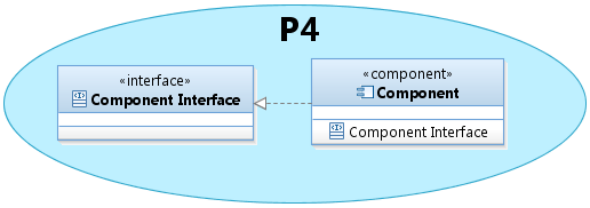

Figure 3. Example of a simple modeling pattern: component and interface

Figure 3 presents 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 attributes, subcomponents and additional interfaces.

III. MAPPING BETWEEN METHODOLOGY AND PATTERN USAGE

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 and further elaboration can be done very early in the project. These patterns will be classified as functional requirements, non-functional requirements, business processes, or business use cases. Idea is to have these patterns and related transformations ready for use in the modeling library that is used for the project. 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 are guidance for choosing patterns that will 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 programmers. Pattern instances in the PSM are mostly implementation of pattern instances in the PIM. For example, in the Component Based Modeling (CBM), the PSM contains platform specific implementations of components defined in the PIM.

As the design of an information system advances through the project, designers can create new pattern instances or elaborate existing ones, as presented in Figure 4. A new pattern instance can be created to document business need, reflect already existing functionality that will be reused, or by transforming from already existing pattern instance in the modeling space. Transformation between pattern instances will probably be 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

In the MDA specification [1], various different model-to-model transformation examples can be found. Transformation can be done within the same model, between two different models, for model aggregation, or model separation. Grunske et al. [6] are presenting 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

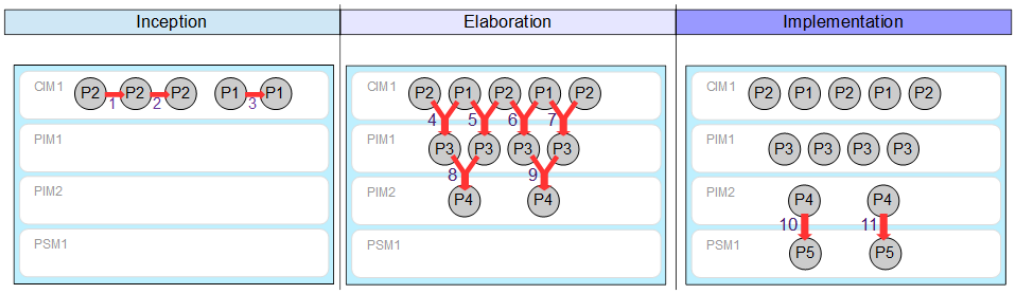

Figure 4. RUP and advancement through a design of an information system

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 is the procedure for translating source model into target model. A modeling space can be defined as a finite set of models $M_S = \{M_1, M_2, \dots, M_n\}$. Each model is a finite set of artifacts $M_i = \{a_1, a_2, \dots, a_m\}$. A transformation is a function $tr: M_S \rightarrow M_S$ that takes a set of artifacts ar_{So} from a set of source models $So \subseteq M_S$ such that $ar_{So} \subseteq \bigcup So$, analyses this set of artifacts and translates them into another set of artifacts ar_{Ta} in a set of target models $Ta \subseteq M_S$, such that $ar_{Ta} \subseteq \bigcup Ta$. 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 same model M_i on source and target side, or $M_i \in So \wedge M_i \in Ta$. A transformation can use the same source and target artifacts, meaning that $ar_{So} \cap ar_{Ta} \neq \emptyset$ when $So \cap Ta \neq \emptyset$, or it can use two disjunctive sets of artifacts $ar_{So} \cap ar_{Ta} = \emptyset$.

From a pattern point of view, each pattern instance is a set of model artifacts. 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 $M_P = \{P_i: 0 < i \leq m \wedge P_i \subseteq \bigcup M_S\}$. In this context, transformation is a function $tr: M_P \rightarrow M_p$. Such transformation takes a set of source pattern instances $p_{So} \subseteq M_p$, analyses all artifacts in these instances and translates them into artifacts that form a set of target pattern instances $p_{Ta} \subseteq M_p$. Figure 5 shows an example of transformation application to a cross model pattern instance.

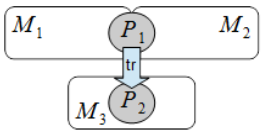

Figure 5. Cross model pattern instance and transformation

Every transformation can be encapsulated in a black box implementation. Such an approach is used in [7] along with the QVT specification. However, taking a step back and observing the QVT specification as one of the transformation approaches, every transformation can be defined as a black box having an interface that depends on the context of transformation usage.

A. Transformation rules

Czarnecki and Helsen [4] are giving important features of transformation rules. Since a transformation can be implemented in many different ways, we must observe it in more abstract and generic way. No matter if we use a declarative or imperative approach, each transformation is a set of rules that creates a relationship between a set of source artifacts and a set of target artifacts. Features and principles given in [4] can be applied to these transformation rules.

A transformation written in QVT-R [7] 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 artifacts in target pattern instances, in order to reflect all the details found in source pattern instances.

1) Validation of pattern instances and imposing constraints

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

As the first step, transformation must validate that supplied source pattern instances are matching expected source side of the transformation. A set of mandatory transformation rules must validate source pattern instances. If all mandatory transformation rules are satisfied from the source side then the transformation can be applied to a supplied source, i.e., the transformation can be applied to the source pattern instance that contains all artifacts needed by the mandatory transformation rules.

The second step is the creation of the target pattern instances. Transformation rules must create all target pattern instances and their artifacts. Every pattern is characterized by the mandatory artifacts that define the essence of the pattern, or what makes this pattern different from other patterns. Not all artifacts created by the transformation must be considered for mandatory. Mandatory artifacts in the target pattern instances are created by the mandatory transformation rules. However, not all mandatory transformation rules must create mandatory artifacts in the target pattern instances.

The last step is to create a set of constraints that will disallow designers to change some of the artifacts in the involved pattern instances. Transformation binds involved pattern instances together by imposing constraints on their artifacts. 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 artifacts. Obviously, all mandatory artifacts must be constrained.
2. Breaking transformation binding by changing artifacts that are satisfying source and target side of the mandatory transformation rules. In this case, constrained artifacts do not need to be mandatory.

If we observe a target pattern instance made of l artifacts $P_i = \{a_1, a_2, \dots, a_l\}$, a subset $MP_i \subseteq P_i$ is considered for a set of mandatory artifacts of the pattern instance P_i. If we have a finite set of applied transformations $TP_i = \{tr_1, tr_2, \dots, tr_k\}$ having P_i as an involved pattern instance, we can derive a mapping function $C: TP_i \rightarrow X$, where $X \subseteq P_i$ is a set of artifacts in P_i constrained by a transformation $tr_x \in TP_i$. In the context of the previous definition about

difference between mandatory and constrained artifacts, we can conclude that P_i can be in a situation where $C(tr_j) \cap C(tr_k) = \emptyset \wedge j \neq k$, and $C(tr_j) = MP_i \wedge C(tr_k) \cap MP_i = \emptyset$. Finally, $MP_i \subseteq \bigcup_{j=1}^{k} C(tr_j)$ means that not all constrained artifacts need to be mandatory, but all mandatory artifacts are constrained since we want to preserve a pattern definition.

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 [9]: 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.

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 a transformation, the source pattern instance can be elaborated by adding new details and artifacts. A transformation can be made so that these newly added details automatically update target pattern instances. Artifacts 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 [7] and [13]. While transformation might constrain changes of some artifacts in target pattern instances, changes of unconstrained artifacts 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, or they can be created by using a transformation. 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. However, these constraints must allow elaboration of newly created top-level pattern instances in order to allow adding needed details. Functional or non-functional requirements are typical examples of top-level patterns. An external service definition is another example of such pattern.

In the example in Figure 6, pattern instance P_1 is made of *CodebookComponent* and related interface. All mandatory artifacts are marked with red color. Artifacts added in the elaboration of P_1 are marked with brown color. Mandatory artifacts within P_1 are all we need to declare a component. Transformation tr_1 mandatory rules are responsible for translation of mandatory artifacts from P_1 to P_3. The artifact

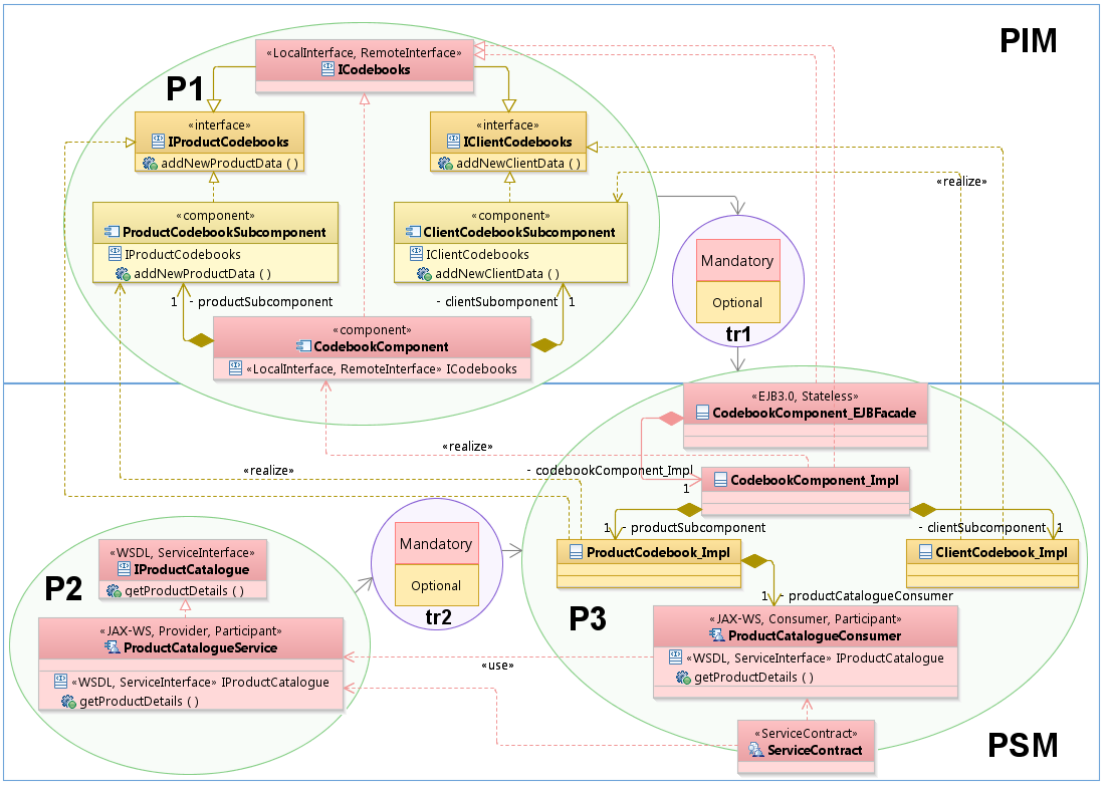

Figure 6. Example of pattern instance transformation

created within P_3 is EJB3 facade as realization of *CodebookComponent*. Transformation tr_1 also created relationships between mandatory artifacts of P_3 and P_1 pattern instances. An information system designer additionally elaborated P_1 and added *ProductCodebookSubcomponent* and *ClientCodebookSubcomponent* together with related interfaces. Transformation tr_1 optional rules translated these subcomponents from P_1 into classes of P_3 and created new relationships between optional artifacts of P_3 and P_1. P_1 and P_3 are bound with transaction tr_1. It means that mandatory artifacts of P_3 are constrained by tr_1 and cannot be changed unless corresponding artifacts in P_1 are changed. Introducing relationships between P_3 and P_1 , such as interface realization, simplifies the propagation of interface changes between these two pattern instances. Propagation of interface changes is a matter of transformation tr_1!

The information system designer's final decision was to reuse the existing service to read product data from a product catalog information system. P_2 is the pattern instance that represents the product catalog service provider. This pattern instance can be created by another transformation from WSDL. Transformation tr_2 is used to translate mandatory artifacts from P_2 representing the service provider, into the set of artifacts for P_3 representing the product catalog service consumer. P_3 must be further elaborated in order to connect *CodebookComponent* realization with the product catalog service consumer.

4) Transformation applicability

As already defined, a transformation takes a set of modeling space artifacts and translates them into another set of artifacts. Earlier definition shows that the transformation can include pattern instances as artifact containers. The size of a pattern instance can be one artifact, up to a whole model. A pattern instance can also be a set of artifacts coming from different models within the modeling space. In order to use transformation, source side of it must be satisfied. Precisely, mandatory transformation rules source side must be satisfied in order for the transformation to be able to create a set of target artifacts and impose constraints on them. If the transformation is applied to a set of pattern instances and a set of source pattern instances satisfies source side of the mandatory transformation rules, the transformation is applicable to this set of pattern instances.

Transformation and related transformation rules, especially if they are written in a declarative programming language such as QVT-R, are logic programs [14]. Transformation tr can be defined as a logic program P, comprised of mandatory and optional set of rules on source and target side. Applicability of a transformation can be derived only from source mandatory rules. If we take a finite set of the mandatory source rules $MSR = \{msr_1(X), msr_2(X), \dots, msr_n(X)\}$, where $X = p_{So}$ is a set of terms, then the applicability of the transformation can be expressed as $\mathrm{A(X)} \leftarrow msr_1(\mathrm{X}) \wedge msr_2(\mathrm{X}) \wedge \dots \wedge msr_n(\mathrm{X})$. Each mandatory source rule is comprised of atoms for checking artifacts within a source pattern instance set $msr_i(X) \leftarrow a_1(y_1, X) \wedge a_2(y_2, X) \wedge \dots a_m(y_m, X)$, where $y_i \in X$. The applicability defined this way can only determine whether a transformation can be applied to a set of source pattern instances or not. Another way is to define a measure of the applicability by expressing percentage of mandatory transformation rules that are satisfied. A finite set of satisfied rules is $MSR_S = \{r(X) \in MSR: r(X) = true\} \subseteq MSR$. The measure of the applicability can be defined as $A_m = |MSR_S|/|MSR| * 100$, or percentage of satisfied mandatory source transformation rules. This measure can help a designer to see which transformations in the modeling library are close to being applicable and what are the differences. Consulting the measure of the transformation applicability is one aspect of the design guidance.

Validation of involved pattern instances is a similar concept to the applicability of the transformation, but it must involve both source and target pattern instances.

V. TRANSFORMATION AND TRACING LANGUAGE

Relationship between model artifacts 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 "pattern instance" defined earlier in this article. Also, application of transformation and imposing constraints on target pattern instances must leave some trail. Creation of a Transformation and Tracing Model (TTM), automatically or manually, can help to resolve before mentioned issues. Every time a new pattern instance is created, new artifact is added into TTM representing this pattern instance. All model artifacts belonging to this pattern instance are automatically bound to it. It can be the result of the transformation, or it can be done manually meaning that a 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 bound to the transformation that created them, since these constraints are the result of the transformation. In order to do this modeling, 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 [13]. Therefore, the TTL must be a MOF metamodel. An overview of the TTL is presented in Figure 7.

 ISBN: 978-1-61208-343-8

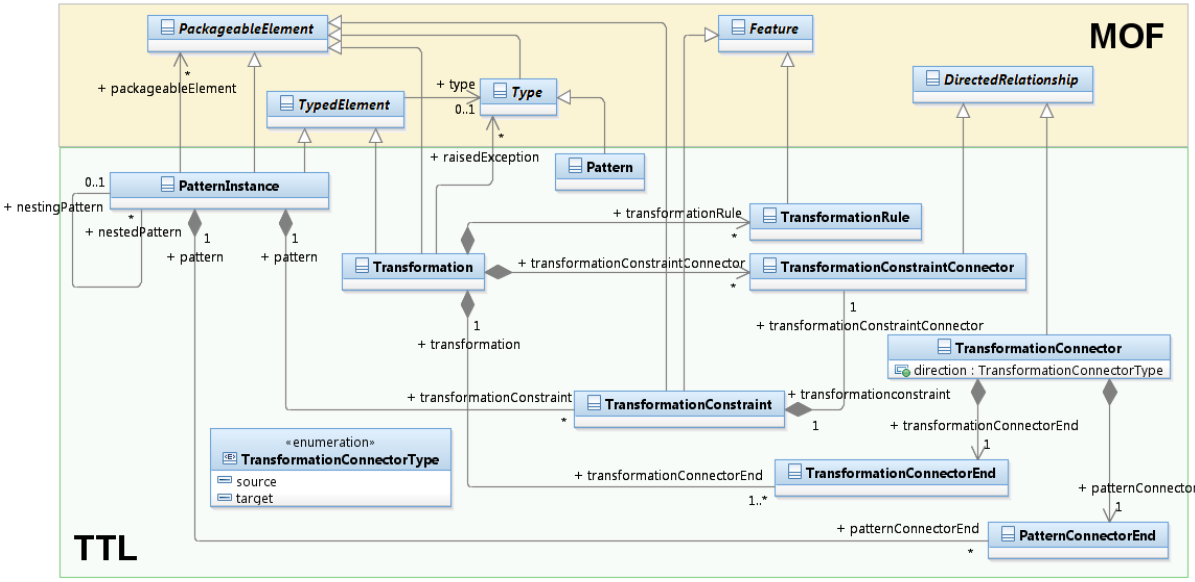

Figure 7. TTL definition

The TTL is having the following elements:

- *Pattern* - A pattern type. Allows classification of pattern instances.
- *PatternInstance* - An element similar to the UML *Package* element. Represents a container for model artifacts. 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.
- *Transformation* - An element defined by its name and type, representing applied transformation. It contains transformation rules used in the transformation. 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.
- *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.
- *TransformationConstraint* - An element defined by its name, representing a constraint on members of a pattern instance imposed by used transformation. This element is contained by the pattern 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.
- *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 8, brown artifacts were created before $tr1$ was applied. We can say that pattern instances $p1$ and $p2$ were designed manually. Green artifacts are produced by the transformation $tr1$. Actions taken during an information system design are automatically stored in 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. GUIDANCE

So far, this article gave only insights into elements needed to establish the method for situational information system design guidance. How to explain designers what is preferred designing practice and how an information system design should look alike? Many companies have well established design practices, from methodology, project activities and modeling point of view. Selection of architectures, technology and practical experience gives a

 ISBN: 978-1-61208-343-8

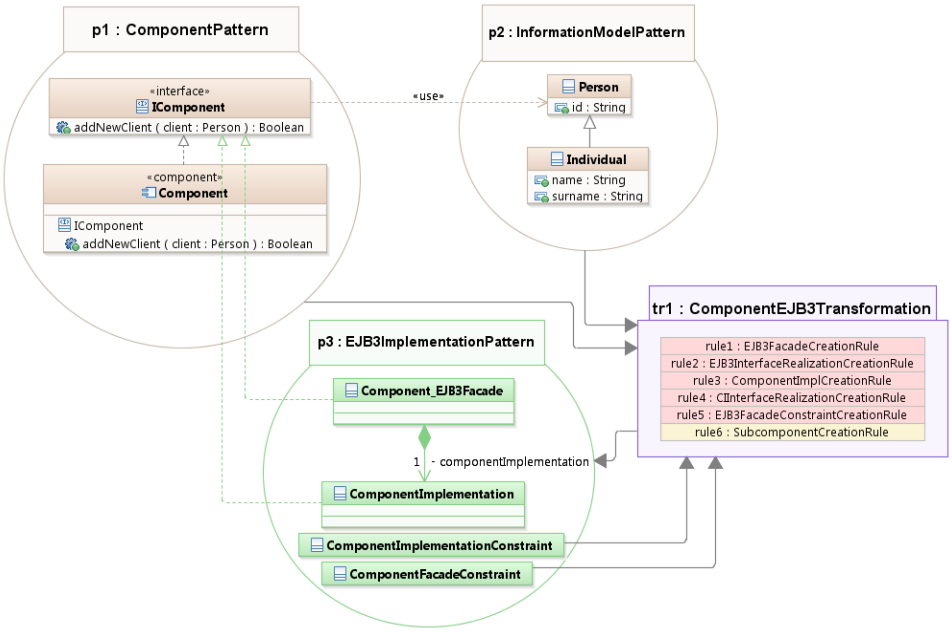

Figure 8. TTM example

company starting point. The method proposed in this article simply takes this experience and allows the company to document their design practices within a modeling tool.

A. *Guidance given through a modeling library*

We already mentioned that a modeling library is comprised of patterns and transformations. Since transformation binds two patterns instances together (as described in the section III), selection of a transformation imposes a selection of involved patterns. Similarly, selection of patterns imposes a selection of potentially applicable transformations.

Applicability and measure of applicability are important transformation features that can be used to give guidance. A designer can elaborate a model or 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 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 rules in the transformation are not correct, meaning that the transformation will never become applicable, this particular transformation can be changed as part of company's design practice evolution.

As already stated before, some transformations can be used exclusively to create new top-level pattern instances. Such transformations are used to create mandatory and optional artifacts in the target pattern instances. Optional artifacts initially created in the target pattern instance can fulfill requirements for the next transformation to become applicable. Further elaboration of this pattern instance can add needed details. It means that applying a transformation can result in a chain of transformations and creation of new pattern instances if a modeling tool is permitted to execute applicable transformations automatically.

Another way is a selection of transformations from the modeling library used in the project. A design lead can manage a set of allowed transformations for his project, limiting designer's choice of applicable or nearly applicable transformations. For example, the architectural decision to use JAX-WS web services will influence the choice of transformations for the project. Similarly, the design lead can manage a set of allowed patterns implicitly by imposing 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 an 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 the 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 ongoing project.

VII. CONCLUSION AND FUTURE WORK

MDA is having two major practical problems: designers have too much freedom while creating information system design models and transformation scope can be very

 ISBN: 978-1-61208-343-8

ambiguous. Usage of a pattern as the main building block for an information system design is a well known approach. In the context of this article, design of an information system is done block by block, allowing the design lead to choose blocks to be used. Such approach allows the design team to use past positive experience to select or define best patterns for the information system they are designing. Another very important element of this method is the usage of transformations to create pattern instances. Transformations must be perceived as the behavioral part of the method. Applicability and measure of the applicability are very important features of the transformation given in this article. They enable controlled application of transformations, which represents guidance for the design team.

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 artifacts of bound pattern instances synchronized. Bidirectionality feature of the transformation helps to reflect changes in both directions. Chain of pattern instances can be easily updated through transformations used to form the chain. Since a pattern instance is supposed to have significantly smaller scope than a model, keeping several pattern instances synchronized during elaboration should be much easier than with big models.

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.

This article is giving only the main idea that can be significantly improved and extended. There are still opportunities for improvement of the mapping between proposed method and methodologies. Also, 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 TMM 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] OMG, MDA. "Guide, Version 1.0.1, 2003." Object Management Group.

[2] 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.

[3] 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.

[4] 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.

[5] OMG, "Core Specification, Version 2.4.1, 2011." Object Management Group.

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

[7] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), Version 1.1, 2011." Object Management Group.

[8] 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.

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

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

[11] 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.

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

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

[14] 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, Jul. 1991, pp. 619-649, doi: 10.1145/116825.116838.

[15] 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.

 ISBN: 978-1-61208-343-8