Electronic Communications of the EASST
Volume 31 (2010)

Proceedings of the
Second International Workshop on
Visual Formalisms for Patterns
(VFfP 2010)

A Generic Technique for Domain-Specific Visual Language Model Refactoring to Patterns

Karen Li, John Hosking, and John Grundy

12 Pages

Guest Editors: Paolo Bottoni, Esther Guerra, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Generic Technique for Domain-Specific Visual Language Model Refactoring to Patterns

Karen Li[1], John Hosking[1], and John Grundy[2]

[1] {k.li, j.hosking}@auckland.ac.nz
Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand

[2] jgrundy@swin.edu.au
[2]Faculty of Information and Communication Technologies, Swinburne University of Technology, PO Box 218, Hawthorn, Victoria, Australia

Abstract: As the popularity of domain-specific visual languages (DSVLs) grows, concerns have arisen regarding quality assurance and evolvability of their meta-models and model instances. In this paper we address aspects of automated DSVL model instance modification for quality improvement based on refactoring specifications. We propose a graph transformation-based visual language approach for DSVL authors to specify the matching and discovery of DSVL "bad model smells" and the application of pattern-based solutions in a DSVL meta-tool. As an outcome, DSVL users are provided with pattern-based design evolution support as refactorings for their DSVL-based domain models.

Keywords: Meta-tools, domain-specific visual languages, graph transformation, design patterns, model refactoring, model-driven engineering

1 Introduction

As the popularity of DSVLs grows, concerns have arisen over the quality of both DSVL designs and the domain models created by novice users using them [Moo09, LB05]. Model quality assurance research is immature, with limited outcomes in the areas of model measures, metrics, and transformations [RB09]. In this context, our research aims to enable DSVL authors to specify predictable model quality problems for detection and correction in DSVL model instances. This is supported at the same time and level as specification of a DSVL. We aim to enable cross-DSVL reuse of common pitfalls and solutions to ease both the DSVL specification burden and to improve the quality of domain models created by DSVL end users.

Refactoring [FB99, Ker05] is a mature technique integrated in most popular IDEs for evolutionary code design, allowing identification of "bad code smells": such as poor naming, unnecessary code duplication and over-complexity. These code problems are then addressed by using "best practise" solutions, typically sets of design patterns that offer tried-and-tested solutions, to improve code design quality [Ker05]. We propose that such a refactoring approach is desirable to improve model quality at higher levels of abstraction by removing "bad model smells", such as unnecessary model element duplication, over-complexity, poor naming and layout, poor relationships, redundancy, incompleteness and inconsistency. Application of appropriate modelling patterns to address these pitfalls would improve model

quality. As in code refactoring IDEs, automated support for refactoring domain models is desirable.

However, to date very few modelling tools provide integrated automatic support for detecting bad model smells and invoking appropriate model refactoring techniques. The state-of-the-art only supports very limited types of models (mainly just UML) with pre-defined, hard-coded refactoring methods. These currently lack a generic way of expressing common but customisable smells and their linked refactoring solutions in DSVL tools [MTM07, MRG09]. Meta-tools are an approach for specifying DSVLs via meta-modelling and generating DSVL environments from specifications. Example meta-tools include MetaEdit+ [KLR96], Marama [GHHL08], and Microsoft DSL Tools [Mic08]. We see meta-tools as a suitable platform to integrate refactoring specifications. Here meta-model level definition of pattern matching and refactoring rules can be integrated as a behavioural extension to the DSVL meta-model. Bad model smell detection and model refactoring support can then be generated from the high level specifications in a similar manner to the modelling support features of the target DSVL tools.

[MMBJ09] outlines a set of problems for reusing refactoring specifications across different meta-models, including: differing element names and types, relationships and roles; and layout and appearance impacts. One technique for customisation of such generic refactoring specifications uses meta-model attributes or parameters for domain context binding and graph patterns or forms (decouple generic pattern-based meta-model and language concept, and facilitate their integration) [GLD08, ZLG05]. Ali's critic tool is similar, but provides a visual approach [AHHG09]. Another technique uses model typing and aspect weaving adaptations: generic meta-model and pattern specifications are defined, followed by adaptations of target domain meta-models to obtain conforming properties applicable to generic specifications [MMBJ09, ZLG05]. This requires effort from users to develop generic typing and aspect definitions (usually OCL-based). In the above, hidden dependency (e.g. one domain element involved in multiple pattern roles; multiple domain elements share the same pattern role) and visibility (e.g. display of both domain contexts and generic pattern elements, plus pattern participation bindings) are unresolved problems. We believe a sound refactoring technique preserves original domain meta-models, using them as the basis for behavioural extension, while supporting visualisation of separate juxtaposed generic specifications and domain configurations to mitigate reuse and visualisation problems. DSVL refactorings may also need to consider surface notation and layout issues that code and meta-model refactorings may not.

Our research aims to generalise a family of common bad model smells (antipatterns) and pattern solutions to improve DSVL modelling. We support generic, customisable refactoring specifications for model-driven reuse across different DSVL meta-model definitions in a meta-tool. We describe integration of a graph transformation-based visual language technique into a DSVL meta-tool for pattern-based DSVL model refactoring specification.

2 Common DSVL Model Refactorings

To better illustrate our intent, we describe several pairs of model refactorings and their generic aspects for reuse potential. For each model example pair, we identify the commonality of their bad smell and refactoring solution.

2.1. Extract duplicate relation

Example 2.1.1 illustrates the refactoring of a UML model, which involves Classes (e.g. Wheel and Car) and Composition relationships (e.g. A Car composes one-to-many Wheels). The bad smell is a duplicate Composition relationship, and a refactoring solution to address this problem is to extract the Composition to a super Class. Example 2.1.2 shows a similar bad smell in a different UML model instance, but with regard to an Association relationship. The refactoring solution is similar but deals with extracting the Association to a super Class.

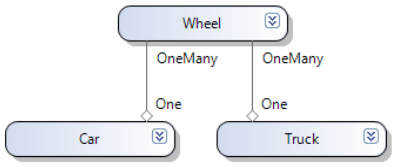
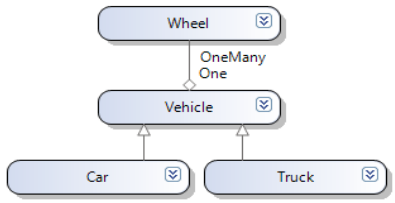
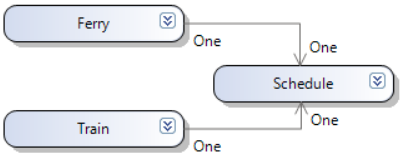
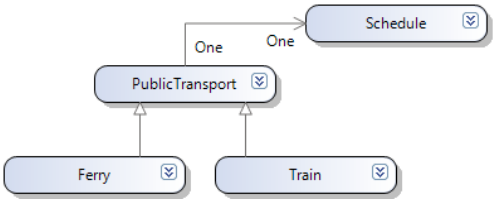
	Before refactoring	*After refactoring*
2.1.1. Extract Composition refactoring in a UML model	 **Bad smell:** a duplicate Composition relationship holds from two source Classes to one target Class	 **Refactoring solution:** extract the Composition relationship for presence between a new super Class and the target Class
2.1.2. Extract Association refactoring in a UML model	 **Bad smell:** a duplicate Association relationship holds from two source Classes to one target Class	 **Refactoring solution:** extract the Association relationship for presence between a new super Class and the target Class

Table 1. Generic Remove Duplicate Relation refactoring used in UML models

Extract Composition and Extract Association, can be generalised as a generic Extract Duplicate Relation refactoring pattern, with a generic bad smell: a duplicate relation between two source participants and a target participant in a domain model. A generic refactoring solution extracts the relation for use between a new parent participant of the two sources and the target. This generic articulation can be specialised to the two refactorings above and others.

2.2. Pull up common element

Example 2.2.1 is another UML refactoring. The bad smell is a common attribute between two sub-classes, resolved by pulling the attribute up to the super class. In Example 2.2.2 a Web Service Composition DSVL model, comprises a composite Service (Enrolment Service), two sub Services (Student Service and Administration Service), and a set of service Operations (e.g. Login, Apply Enrolment). A similar refactoring pulls up the common Operation from the sub Services to the composite Service. These two examples can be generalised to a generic Pull Up Common Element refactoring pattern.

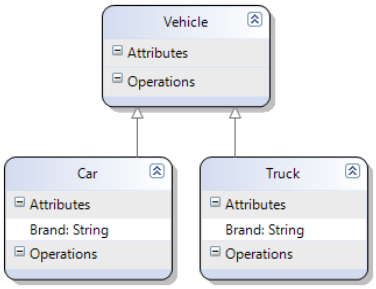
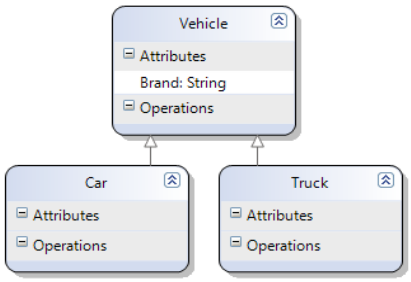
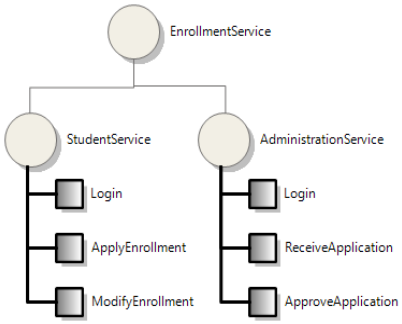
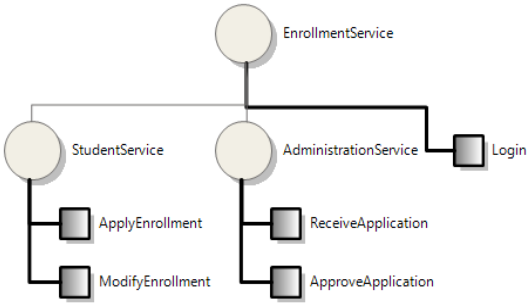
	Before refactoring	*After refactoring*
2.2.1. Pull Up Common Attribute refactoring in a UML model	 **Bad smell:** a common Attribute (with the same name and type) holds for two Classes	 **Refactoring solution:** pull the Attribute up in the super Class
2.2.2. Pull Up Common Operation in a Web Service Composition model	 **Bad smell:** a common Operation (with the same name) holds for two sub Services	 **Refactoring solution:** pull the Operation up in the composite Service

Table 2. Generic Pull Up Common Element refactoring used in UML and Web Service Compostion models

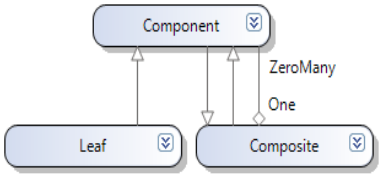
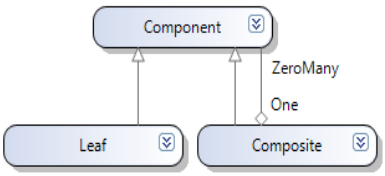
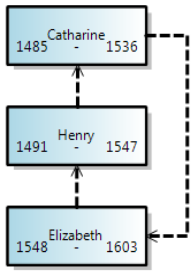
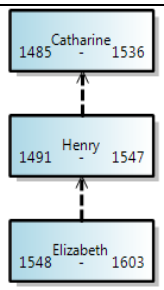
	Before refactoring	*After refactoring*
2.3.1. Remove Circular Inheritance refactoring in a UML model	 **Bad smell:** circular Inheritance relations hold between two Classes	 **Refactoring solution:** remove the last added Inheritance relation
2.3.2. Remove Circular ParentRelation in a Family Tree model	 **Bad smell:** circular ParentRelation links hold among a chain of Persons	 **Refactoring solution:** remove the last added ParentRelaion link that caused the circularity

Table 3. Generic Remove Circular Reference refactoring used in UML and Family Tree models

2.3. Remove circular reference

Example 2.3.1 shows a bad smell of circular Inheritance relations between two UML classes (Component and Composite). A refactoring is to remove the last added Inheritance relation that caused the circularity. A similar refactoring in a Family Tree model is shown in Example 2.3.2, to resolve circular ParentRelations between two Persons (Catharine and Elizabeth).

2.4. DSVL model refactoring pattern library

We have identified a range of common refactorings applicable across DSVLs such as those above. We are developing a pattern catalogue with a growing number of generic refactoring patterns accessible by DSVL authors in a DSVL meta-tool. Each refactoring pattern is represented in a generic visual language described below. DSVL authors can contribute new patterns, and with an ongoing extension, the catalogue can be analysed for overlaps (e.g. identical smells or solutions in different refactoring specifications) and conflicts (e.g. opposite refactoring solutions for identical smells) at the specification time. Applying chosen refactoring solutions in the target DSVL tool needs to consider not just model instance update but diagram update, possibly including layout change, update of different representations in different diagrams, and update of multiple diagrams showing the refactored items.

3 Visual Specification of Model Refactoring

Various formalisms have been used to specify model refactoring [MTM07]. One we are convinced is appropriate is graph transformation (rule-based modification of graphs) [Roz97]. This is because it presents an intuitive graphical computation paradigm and a natural fit for describing matching of bad model smells. It also empowers effective validations of specifications through parsing graph grammars. In this approach the left-hand side (LHS) model (source) and right-hand side (RHS) pattern solutions (target) correspond well to source and target of transformation rules in our domain. We believe UML-based approaches lack a natural visual linkage between bad smells and refactoring solutions.

Specification of anti-patterns and patterns using graph transformations to support model evolution is an existing technique [BEK+06, ZKDZ07, GLD08]. However, current solutions don't separate domain meta-model contexts from common transformation specifications. While some pattern-based reuse within the specified domain is facilitated, reuse across different DSVLs is not [MTM07]. Our approach aims to provide such a separation by using layers and via a generic but configurable visual language. Our approach allows DSVL designers to define, with high-level reuse support, refactoring of bad model smells at the same level of abstraction as their DSVL meta-models. The DSVL meta-models are the domain profiles and generic refactoring specifications are customisable using profile stereotypes.

3.1. Specifying refactoring at the DSVL meta-model level

In our approach refactoring is specified at the meta-model level using a linked view to a DSVL meta-model designer that includes elements such as domain classes, relationships, shapes, connectors and diagram element maps. Generic refactoring specifications are to be contextualised with such structural meta-model elements. The view exploits parallel

orthogonal layered representations, as shown in Figure 1, for separate but easy to bind generic refactoring pattern specifications in the top layer (a) and DSVL meta-model contexts in the bottom layer (b). This achieves genericity, customisability and reusability. Form-based filtering capabilities support adding in interested potential DSVL meta-model elements for pattern participation in the lower domain model layer. Customisations of generic refactorings are to be specified via visual cross-layer links. We elaborate the visual notation as follows.

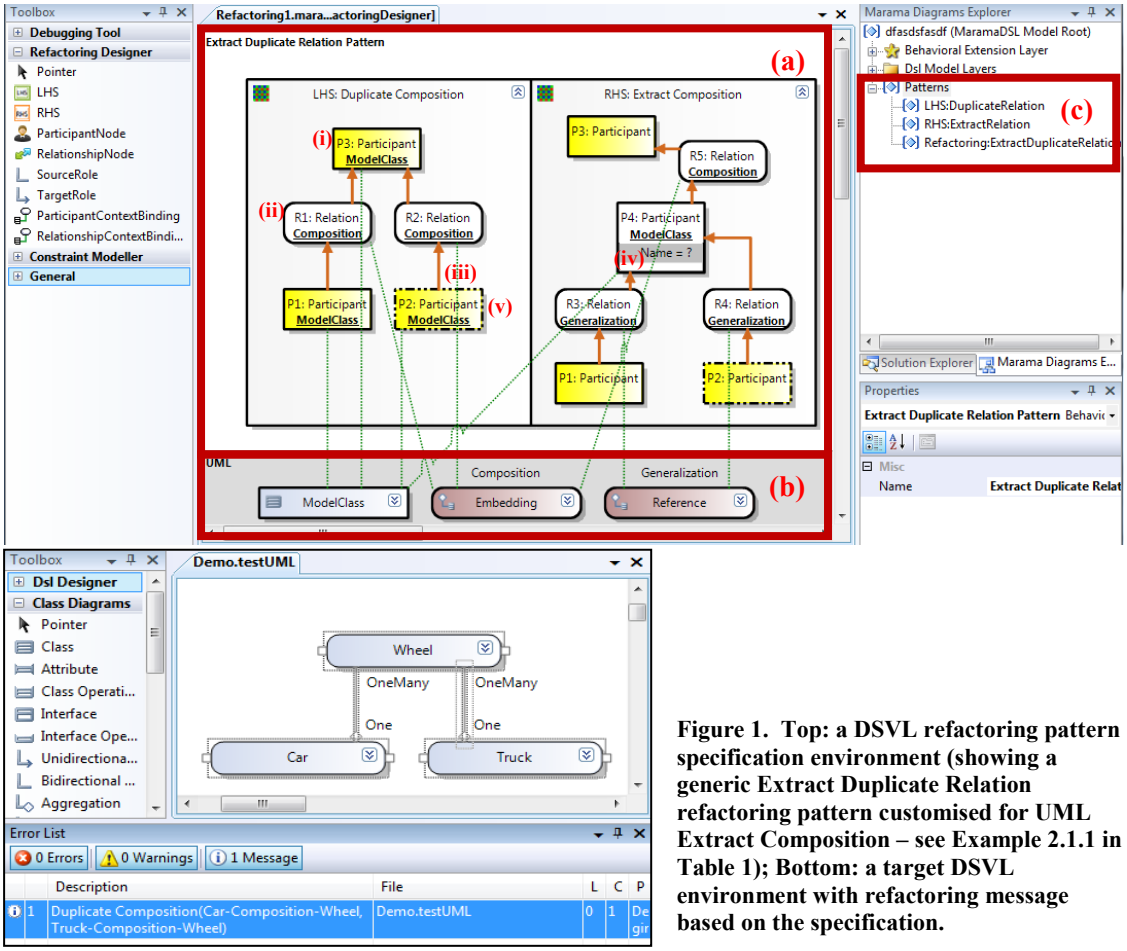


Figure 1. Top: a DSVL refactoring pattern specification environment (showing a generic Extract Duplicate Relation refactoring pattern customised for UML Extract Composition – see Example 2.1.1 in Table 1); Bottom: a target DSVL environment with refactoring message based on the specification.

3.2. Generic notation in graph transformation paradigm

Our refactoring specifications contain two parts, shown in Figure 1 (a). A bad model smell (transformation precondition) is LHS and pattern solution (post-condition) RHS of a graph transformation rule. This effectively specifies when to apply a refactoring (LHS) to the consequence of applying it (RHS). Both the LHS and RHS use the same node, edge and attribute notations, with nodes specifying participants and relationships, edges specifying role bindings, and attributes specifying additional property-based condition checking criteria (e.g. property pattern-matching conditions, local or global constraints) or input acquisition (e.g. user prompt, or calculated dependent value). LHS to RHS mappings define the transformation, and are encoded using identical naming, numbering and colouring. Mapped constructs represent

preserved model element structure, unmapped LHS constructs are deleted, unmapped RHS constructs are created, and attributes of mapped constructs represent updates.

Our visual language defines the following basic notational elements (also shown in Figure 1) for a generic DSVL refactoring specification:

1. Generic participant nodes (i), represented by rectangular compartment shapes (holding attribute specifications, collapsed by default) with labels encoding identification number and name, and a placeholder for a to-be bound DSVL meta-model context;
2. Generic participant relationship nodes (ii), represented by rounded rectangular compartment shapes with labels encoding identification number and name, and a placeholder for a to-be bound DSVL meta-model relationship context;
3. Edges (iii), are directed connectors between participants and relationships representing source and target role bindings;
4. Attributes (iv), as compartment members of a participant or relationship, specify pattern matching conditions as a mechanism to formulate refactoring rule application conditions as global or local graph constraints. They are currently specified using C# expressions, which we intend replacing by simplified visual OCL expressions as per our earlier DSVL constraint specification mechanism [LHG07].

Building on this base representation explained above, we also enrich the node and edge representations with border line styles to explicitly express the following crucial factors:

1. Scaling up pattern matching horizontally. Consider Example 2.1.1 again, we want to catch and refactor the same bad smell when an arbitrary number (>2) of source classes are present, each related by Composition to the same target. The default pattern participant/relationship node in our language captures only one matched instance element. We use a dashed border on a participant (e.g. Figure 1 (v)) to represent an elision of a number of horizontally like participants (i.e. multiple sibling instances under the same relationship). Horizontal scaling-up is clearly distinctive from vertical scaling-up which is represented on a relationship as explained next.
2. Scaling up pattern matching vertically. As seen in Example 2.3.1 and 2.3.2, a generic Remove Circular Reference refactoring specification should be able to capture the circularity no matter when it presents between two immediate participants, or in a similar role chain of an arbitrary number of participants. We use dashed borders on a relationship and its source and target role edges (e.g. Figure 2 (i)), to represent an elision of a role chain with a number of like participants under the same relationship type guardian.
3. Imposing implicit (queried) conditions. Our node notation has attribute compartment fields for specifying dynamically queried conditional characteristics for pattern matching. For instance, an attribute in a participant may be specified to hold a certain data value; an attribute in a relationship may specify equality of certain data values between related participants (e.g. Figure 3 (ii)). We used a thickened and coloured border to represent such a node (e.g. Figure 3 (i)) with an implicitly queried attribute.

3.3. Configuring with DSVL meta-model elements

The customisation of a generic refactoring specification is a domain context binding process where general abstract pattern elements are instantiated with concrete domain element types.

In our approach the context binding of a DSVL meta-model is visually represented by green dotted lines across the two layers connecting elements in the DSVL meta-model with their participations in the refactoring pattern specification layer. It is a simple cross-cutting linking mechanism. The context binding links can be concealed at individual pattern element levels for diagram clutter management. Context bindings are supplemented by a dual text encoding on a pattern element, via underlined text in the bound pattern element, which can also be concealed by collapsing the pattern element, to ease context navigations.

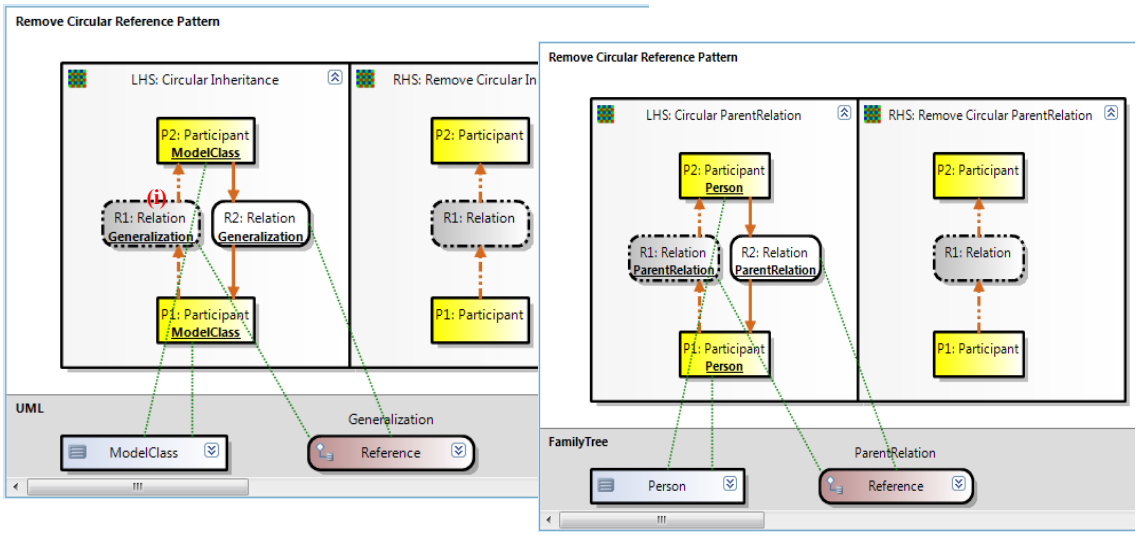


Figure 2. Generic Remove Circular Reference refactoring pattern customised for UML and Family Tree domain models respectively (see Example 2.3.1 and 2.3.2 in Table 3)

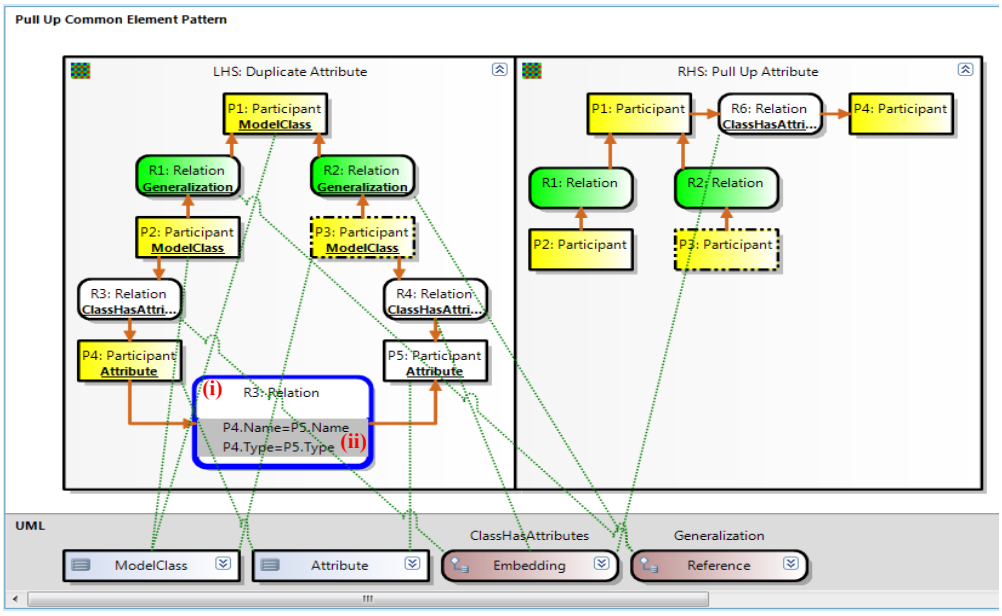


Figure 3. Generic Pull Up Common Element pattern customised for UML Pull up Attribute (see Example 2.2.1 in Table 2)

We elaborate customised refactoring specifications for two examples shown in Section 2. With the context binding links established, the generic Extract Duplicate Relation refactoring pattern specification in Figure 1 is customised for a UML Extract Composition use case (Example 2.1.1 in Table 1). It specialises the two LHS source ModelClass[1] participants (P1 and P2, scaled to represent multiple occurrences), which have duplicate Composition[2] relationships (R1 and R2), with a target ModelClass participant (P3). It defines the RHS as preserving the matched participants (P1, P2 and P3), but creating a new super ModelClass (P4) for the two source ModelClasses (P1 and P2) using Generalization[3] relationships (R3 and R4), and a new Composition relationship (R5) for the super ModelClass (P4) to connect to the target ModelClass (P3). The unpreserved relationships from the LHS, i.e. the duplicate Composition relationships (R1 and R2), are removed during this refactoring.

Figure 3 shows the specification of such a generic Pull Up Common Element refactoring pattern customised for a UML Pull Up Attribute use case (see Example 2.2.1 in Table 2). The matching condition defined on the LHS is that there are Attributes[4] (P4 and P5) in ModelClasses (P2 and P3) that share the same name and type (queried in an implicit relationship R3). The RHS specifies the preserving of all matched participants (P1, P2, P3 and P4) except a duplicate Attribute (P5), and the preserving of the Generalization relationships (R1 and R2). However, a new ClassHasAttribute[5] relationship is to be created for the super ModelClass (P1) to hold the common Attribute (P4).

4 Tool Support for Realisation and Reuse

Our proof-of-concept tool, MaramaDSL, has been developed as an extension to Microsoft DSL Tools [Mic08], a Visual Studio-based meta-tool. Our tool provides linked designer views, framework code and code generators to allow refactoring specifications to be integrated at meta-model level with a DSVL definition in the Microsoft DSL Tools. We have developed an extensible library of functional building blocks to be used in code generation for pattern matching based selection, insertion, deletion and update of model elements.

With our tool support, a refactoring specification generates code for a DSVL environment that informs users of detected bad smells as they occur, and provides commands to apply pattern solutions to model instances. In a target DSVL tool environment with user-created domain models, matched bad smells are shown to the user in a Message window (Figure 1, bottom); double clicking smell messages highlights the pattern participants and relationships in the domain diagram. Right-clicking on the diagram or the message will bring up a context menu command to enable execution of the refactoring rule as defined.

Given the genericity characteristic of our visual language, cross-DSVL reuse can be readily achieved. MaramaDSL provides support for high-level separate and holistic reuse of model bad smell definition, pattern solution specification, and the overall refactoring transformation.

[1] ModelClass is the name of the meta-model element in a testing UML tool representing a class.
[2] Composition is the name of the meta-model element representing a composition relationship.
[3] Generalisation is the name of the meta-model element representing an inheritance relationship.
[4] Attribute is the name of the meta-model element in a testing UML tool representing an attribute.
[5] ClassHasAttribute is the name of the meta-model element in a testing UML tool representing the relationship between a class and an attribute.

It allows a whole specification or the LHS/RHS to be saved context-free (with all context bindings removed) into a pattern catalogue, appearing in an explorer window, as seen in Figure 1 (c). This can then be accessed and drag-dropped from there to a refactoring specification diagram for direct adoption, followed by binding with other DSVL meta-models. Accessed pattern specifications can also be easily adapted for reuse in a variant way, e.g. modify or remove any existing participant or relationship, or add elements to meet specific needs.

Figure 2 shows a generic Remove Circular Reference refactoring pattern customised for the UML and Family Tree domain models respectively (see example 2.3.1 and 2.3.2 in Table 3). The same generic refactoring specification is applied directly to both domain models, with the only difference being context binding of the UML and Family Tree meta-models respectively.

We see our concept is general-purpose and can be similarly implemented in other meta-tools such as Marama [GHHL08].

5 Discussion

The aim of this work is to empower DSVL authors with easy-to-use definition of bad model smells detection and pattern solutions with high level model-driven reuse support. We aim to support this at the same level as DSVL specifications and to equip DSVL users with pattern-based evolution support for domain model development. Our realization of this concept is a simple visual language for refactoring specifications which is at the DSVL meta-model level, intuitively using the graph transformation paradigm to link model smells with solutions, and providing common pattern abstraction and configurability for reuse across different DSVLs.

We are conducting usability studies at each stage of the development of our visual language and tool. Subsequent to the preliminary design presented here we have conducted a Cognitive Dimensions [GP96] analysis to evaluate tradeoffs, strengths and weaknesses of our solution. The visual language explicitly models abstract refactoring rule participants and relationships and allow easy configuration across DSVLs through decoupled but interacting layers. It has a clear role collaboration model (*role expressiveness*) specific to refactoring and pattern concepts. It has expressiveness equivalent to domain-specific code written with APIs, with the comprehensiveness of model query and transformation functions (SELECT, INSERT, DELETE and UPDATE), but with a lower *abstraction gradient*, augmented understanding, reduced effort, and a much shallower learning curve via *closeness of mapping* to users' cognitive models of refactoring pattern presentation and use. We have mitigated areas of *hidden dependency* and v*isibility* in the language by *juxtaposition* of orthogonal layered views, and dual coding of custom values through context links and dynamic properties.

Initiating a refactoring specification requires some *hard mental operations* and *premature commitment* when choosing appropriate pattern elements to compose, and understanding overlapping and conflicts in multiple refactorings. However adding *abstractions* in the form of pre-defined patterns in a pattern catalogue reduces complexity and *diffuseness*. The use of the visual language reduces *error proneness* compared to coding, but requires proactive checking of model semantics for correctness. *Progressive evaluation* is allowed but requires a compile-and-run cycle for the generated code. The language uses a *terse* set of graphical symbols but

with a rather *verbose* set of textual labels for expressing pattern elements and domain-specific context bindings. *Diffuseness* caused by that is mitigated by using them within typed symbol groups. We wanted to use layout (e.g. align preserved pattern structure elements in LHS and RHS) as a *secondary notation* as it does not affect any semantics but is good for promoting readability and identification of high-level graph matching patterns. The usual diagram update *viscosity* problems occur i.e. hard-to-change, and require automatic layout support to mitigate.

We are currently focusing on a variety of designs to address the completeness and correctness of the meta-model level refactoring models and to allow multiple specifications to be analysed in order to detect overlapping and conflicts. We aim to provide automated validation of both generic and customised refactoring models. We are looking to integrate an existing graph grammar parser as the backend for validating generic refactoring specifications. We have considered several design options to use visual feedback to inform DSVL authors with missing or conflicting context bindings for customised refactoring specifications. An option we are exploiting is a consistent Message Window-like notification mechanism as per what we used for notifying of bad smells and refactorings in a DSVL model instance.

Multiple refactoring rules may share common bad smell patterns present in the LHSs, which means that multiple transformations may be due to execute at the same time. In some situations executing one will break the condition matched to execute the other. We want to enable DSVL authors to identify overlapping LHSs in multiple specified refactorings and also conflicts if exists, and allow them to set priority order for the detection rules to fire. Allowing this to be identified at DSVL design time removes the burden on DSVL users in determining which refactoring to execute when multiple are due to be executed. Our intent is to provide a high-level visual analysis graph that automatically gathers information from existing refactoring specifications, represents refactorings as nodes with links indicating overlapping LHSs or conflicting RHSs, and allows DSVL authors to set execution orders from there.

To better support DSVL end users with model evolution based on the refactoring specifications, we are also designing dynamic visualisation of pattern matching and refactoring transformation at model instance level with helpful annotation, playback and rollback features.

6 Conclusion

Model refactoring should be generic and reusable across DSVLs, in a similar way that code refactoring has been applied across different programming languages and platforms. We propose adding into a DSVL meta-tool a generic and reusable specification technique for DSVL authors to define model refactorings to support DSVL users evolve their model instances. A graph transformation based visual language approach is proposed for this purpose.

Bibliography

[AHHG09] N.M. Ali, J. Hosking, J. Huh, J. Grundy. Template-based critic authoring for domain-specific visual language tools, In Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2009. pp.111-118, 20-24 Sept. 2009.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. In Proc. of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra2006), 2006.

[FB99] M. Fowler, K. Beck. Refactoring: improving the design of existing code. Reading, MA: Addison-Wesley, 1999.

[GHHL08] J. Grundy, J. Hosking, J. Huh, K. Li. Marama: an Eclipse Meta-toolset for Generating Multi-view Environments, in Proc. of the 30th International Conference on Software Engineering (ICSE'08). 2008: Leipzig, Germany.

[GLD08] E. Guerra, J. d. Lara, P. Diaz. Visual specification of measurements and redesigns for domain specific visual languages. Journal of Visual Languages and Computing 19(3), pp. 399-425, 2008.

[GP96] T.R.G. Green, M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 7, pp. 131-174, 1996.

[Ker05] J. Kerievsky. Refactoring to patterns. Boston: Addison-Wesley, 2005.

[KLR96] S. Kelly, K. Lyytinen, and M. Rossi. Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in Proc. of CAiSE'96. 1996.

[LB05] F. Leung, N. Bolloju. Analyzing the Quality of Domain Models developed by Novice Systems Analysts. In Proc. of the 38th Hawaii International Conference on System Sciences, 2005.

[LHG07] N. Liu, J. Hosking, J. Grundy. MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism. In Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2007.

[Mic08] Microsoft Domain Specific Language Tools. http://msdn.microsoft.com/en-us/vsx/default.aspx, Microsoft, 2008.

[MMBJ09] N. Moha, V. Mahe, O. Barais, J.-M. Jezequel. Generic Model Refactorings. In Proc. of MoDELS, pp. 628-643, 2009.

[Moo09] D.L. Moody. The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE TSE 2009.

[MRG09] M. Mohamed, M. Romdhani, K. Ghedira. Classification of model refactoring approaches. Journal of Object Technology 8(6), 2009.

[MTM07] T. Mens, G. Taentzer, D. Müller. Challenges in Model Refactoring. In Proc. of 1st Workshop on Refactoring Tools, University of Berlin, 2007.

[RB09] J. Rech, C. Bunse. Model-driven software development: integrating quality assurance. Hershey: Information Science Reference, 2009.

[Roz97] G. Rozenberg. Handbook of graph grammars and computing by graph transformation. World Scientific, c1997.

[ZKDZ07] C. Zhao, J. Kong, J. Dong, K. Zhang. Pattern-based design evolution using graph transformation. Journal of Visual Languages and Computing 18(4), pp. 378-398, 2007.

[ZLG05] J. Zhang, Y. Lin, J. Gray: Generic and domain-specific model refactoring using a model transformation engine. Volume II of Research and Practice in Software Engineering, pp. 199-218, 2005.