In Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centgric Computing, Corvallis, OR, USA

Template-based Critic Authoring for Domain-Specific Visual Language Tools

Norhayati Mohd.Ali[1], John Hosking[1], Jun Huh[1] and John Grundy[1, 2]
[1]*Department of Computer Science and* [2]*Department of Electrical and Computer Engineering*
University of Auckland,
Private Bag 92019, Auckland, New Zealand
nmoh044@aucklanduni.ac.nz, {john, jhuh003, john-g}@cs.auckland.ac.nz

Abstract

In recent years, there has been an increasing interest in using computer-based "critic tools" for supporting the end-users or tool designers in analyzing and proactively improving their design artifacts. Several approaches have been applied to designing and realizing such critics, for example rule-based, pattern-matching and knowledge-based approaches. While many studies have found evidence that critic tools are an efficient feedback-providing mechanism, there is still insufficient knowledge about how to provide an effective critic authoring environment. In this paper we apply a business rule template approach as a mechanism to specify and realize critics for Marama-based domain-specific visual language tools. We describe a visual design critic authoring template approach that supports end-users or tool designers in the construction of critics for any Marama-based tools.

1. Introduction

A computer-aided critic tool (also known as a critiquing system) is a tool that provides guidelines, advice or feedback on/for particular design artifacts [17, 8]. The concept of a *critic* is one that has been adopted in various domains, including medical systems, programming, design, education, expert systems, and decision support systems. Many studies have found evidence that proactive critiquing tools are an efficient feedback-providing mechanism. These tools offer several benefits including proactive design improvement, early error detection, and heuristic-based guidance and context-sensitive feedback.

As a simple example consider a software designer manipulating a design artifact in an editing tool. The tool's critics analyze the design artifact as it changes and reveal to the designer some potential problems/ errors with the design artifacts, e.g., wrong naming convention, over-complex design relationships, and potential misuse of design domain concepts. The critic tool will offer feedback, or "critique" the design, usually proactively as the design evolves. The tool may also suggest alternative design decisions to the designer to resolve potential problems. The interaction between designer and critic tool is iterative until the designer is satisfied with the design artifacts. Typically, critic feedback is kept "unobtrusive" to the designer so as not to overly interfere with the design process.

One of the most significant examples of a critic tool is ArgoUML [16] an open source Unified Modeling Language (UML) CASE tool that supports the editing of UML notation diagrams. Its critics offer suggestions to designers when a software architecture diagram violates various UML rules [17]. The LISP-Critic [8], Argo [17], ABCDE-Critic [18], IDEA [4] and RevJava [9], are further examples of critic-based tools in the software design domain. These tools were developed for the domains of LISP programming, software architecture, object-oriented analysis and design, design patterns and Java object-oriented software, respectively. While many studies have reported that critic tools provide an efficient mechanism for feedback, critic authoring continues to be a challenge [15]. There is little agreement on how critics should be specified and little work on tools to author and realize critics in complex design environments.

We present our research on the *Marama Critic Definer*, a critic support-based extension to our Marama metatool. Marama is a metatool implemented as a set of Eclipse [6] plugins that is used for building complex multi-view diagramming applications [11]. Designing and implementing critics in Marama, as with most other tools, has previously required low-level coding. With our critic authoring extension the tool designer, not intended to be a skilled programmer, is able to define his/her own critics and feedback mechanisms specific to their tool. The Marama critic designer automatically generates code to realize the critic in the target tool using Marama's APIs.

In this paper we present the background and motivation for this research. We then present our

approach to a critic authoring mechanism adapting business rule templates. A case study to illustrate the use of MaramaCritic is presented and implementation of MaramaCritic is described. Finally, we discuss the implications of our work and suggest further research.

2. Motivation

Many design tools would benefit from proactive critiquing support. For example, our MaramaMTE architecture design tool [12] needs to support a range of critics to advise on software architecture modeling and performance engineering. These critics can greatly assist the designer, a software architect, in producing high quality designs. MaramaMTE is implemented using our Marama meta-toolset. While this provides high-level abstraction for visual modeling tool development the tool developer must specify such critics using Marama's APIs (i.e., Java code), as are ArgoUML critics [17]. This approach is very time-consuming and difficult and most unsuitable for all except experienced tool platform developers.

Ideally a critic authoring approach would provide high-level abstractions for tool developers to specify: design artifacts of interest; queries over design artifacts indicating problems or potential issues with a design under construction; a way to inform tool end-users of the problem in an unobtrusive way; one or more ways to (semi-)automatically resolve the design problem, if one exists, and to help the designer do this [18, 16].

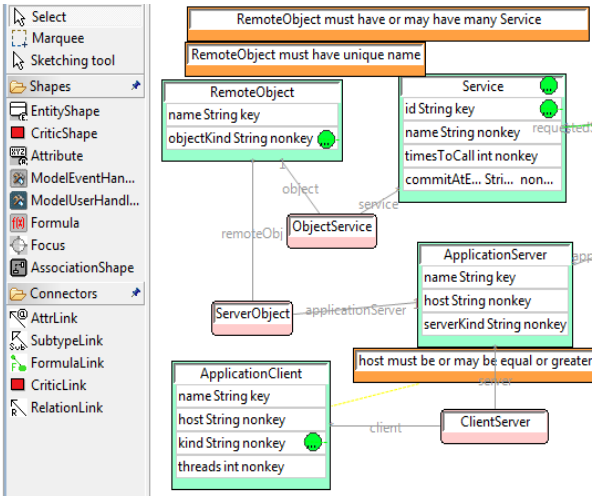

Figure 1. MaramaMTE critics defined using the Marama meta-model editor [1].

In earlier work we developed both taxonomy of critic authoring needs and initial visual critic authoring elements to support the authoring of critics for Marama-based domain-specific visual language tools [1]. In this first attempt, the critics for Marama-based tools were defined using the existing Marama meta-model editor via a functional element, called *Critic,* that was created in the Marama meta-model editor [1, 11]. An example of this approach is shown in Figure 1. Unfortunately this proved to be unscalable and our initial visual critic specification language was unsuitable for tool developers.

Researchers have explored several different approaches to critic authoring. Qiu and Riesbeck [15] equip their Java Critiquer tool with a critique rule authoring facilities that permit the user (teacher) to check and modify critics generated by the tool [15]. Java Critiquer is a critic tool that checks Java statements in a student program code using a pattern matching approach. These patterns are suitable for textual languages but require complex knowledge of regular expressions and language syntax.

Robbins and Redmiles describe ArgoUML [16], a tool for object-oriented design that supports several identified cognitive needs of software designers. Although critics in ArgoUML are coded via Java classes, the users (critic implementers) can still add new critics because ArgoUML provides a class framework, source code templates and examples describing how to create new critics [2]. This requires coding using Argo's Java APIs and thus realistically only programmers can build or modify critics.

Souza et al. [18] present ABCDE-Critic, a system that has a construction kit that supports UML class diagrams, a hypermedia system, and a critic system. ABCDE-Critic allows end-users to define critics via a first-order production system notation.

Bergenti and Poggi [4] introduce the IDEA tool that automatically detects patterns employed in a UML diagram and produces critiques about the pattern usage. Critics in IDEA are realized via a knowledge-based approach and Prolog rules [4]. IDEA lets the user (programmers and architects) add new patterns, new rules, and new critiques because it is extensible and customizable. However, a high-level understanding of design patterns and detailed Prolog knowledge is required.

The type of critic tools summarized above are not similar to our Marama Critic tool, because: 1) the tools are not metamodelling tools, and 2) the task of authoring and defining critics is not based on metamodel elements. However, our Marama Critic is similar to tools such as MetaEdit+ [5], KOGGE [5], and other metamodelling tools. Most existing metamodelling tools allow for specifying constraints, checking consistency and completeness [5], and these are expressed based on metamodel elements. Various approaches can be used, such as OCL or Z expressions, to express the constraints. We imitate the

metamodelling approach but our focus is on critics authoring which is inspired by the critic tools.

While a substantial body of literature has been published on critic tools or critiquing systems, there is less discussion on issues of critic authoring, i.e., allowing end-user and tool designers to customize critic rules. Oh et al. [14] point out that most critic rules are written in advance and that their customization is generally very hard. Many of the critic tools summarized above allow for critic customization but the process of authoring or customizing the critics is not easy [1, 4, 17, 18]. The users have to understand both the tool domain and the critic approach used before designing and realizing critics. These barriers need to be mitigated before widespread use of customizable critic mechanisms is adopted.

3. Our Approach

Our initial attempts to add a visual critic authoring capability to Marama introduced several problems [1]. Adding visual critic design elements to the meta-model view in Marama proved to be un-scalable but also lacked sufficient abstractions for tool developers to leverage. To address some of these problems identified we have refined the prototype development by proposing a *Marama Critic Definer* editor. Figure 2 illustrates this new development approach for critics for Marama-based domain-specific visual language tools.

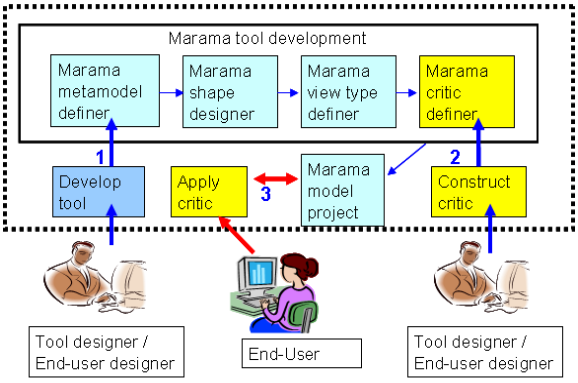

Figure 2. Marama Critic development approach

We have created a new specification tool, the *Marama Critic Definer,* to enable the tool or end-user designer to specify critics for their Marama-based tools. Currently Marama-based tools are defined and developed using three metatool editors: *the metamodel definer*, defining a tool's information model, the *shape designer* view, defining visual notational elements; and the *viewtype definer* view (**1**) [11]. Using our extensions, once the new tool is defined and equipped with sufficient information the tool or end-user designer can select the *critic definer* view to visually author and realize critics (**2**). This task is supported by form-based interfaces, the *critic construction* editor and the *feedback* editor, which allow tool designers to specify critics and feedbacks in a simple, intuitive way. We describe this approach in detail in Section 4.

A tool end-user can then create new modeling projects and diagrams using the new tool, which is generated by Marama as a set of Eclipse plug-ins. When a diagram is created, critics for that particular tool are instantiated. If a user creates design content that a critic identifies as problematic then a critique will be generated to warn the user about the potential problems/errors (**3**). Feedback from the critic is displayed to allow the user to fix the problem/error. Some critics may provide assistance to the user in correcting problems.

4. Critic Authoring Templates

Our approach to supporting the critic-authoring task is to adapt the concept of "business rule templates" to critic authoring. We took this approach because the metamodel for Marama tools is expressed using Extended Entity Relationship (EER) descriptions. This matches the properties defined in "business rule templates". We also chose the business rule templates approach to allow end-users with limited programming capability to define and author critics for software tools.

We have developed a set of reusable design critic templates that apply the Business Rule (BR) templates approach [13, 19]. The BR templates are formal sentence patterns that allow the expression of business process rules usually applied to the domain of business process modeling. Rule templates are applied to business process meta-model elements to constrain or reason about the correctness of target business process model instances. Inspired by the BR templates approach we adapted this concept to apply it to the critic authoring domain, forming a set of reusable critic templates. Our critic authoring templates are applied to a target domain-specific visual language tool's meta-model to constrain and/or reason about its target model instances. Currently our templates consist of two types: *constraint templates* and *action assertion templates*. Constraint templates specify desired or undesired states of models while action assertion templates say what to do when an undesired state is detected (including critique generation and possible resolution actions). Critic authoring templates can be chained together to specify complex patterns over a meta-tool's model instances and complex critique/resolution strategies.

Constraint templates are further divided into two types: attribute constraint and relationship constraint templates. The former are used to specify desired or undesired properties around uniqueness, optionality (null), and value check of an entity's attributes [13]. The latter assert the relationship types, as well as the cardinality and roles of each entity participating in a particular relationship. Chaining a mixture of attribute and relationship templates together allows a tool designer to specify complex detection patterns over their tool meta-model. Action assertion templates specify an action to be activated on the occurrence of a certain event or on the satisfaction of certain conditions. These include critique message generation for the tool user and/or "fix up" operations to apply to resolve detected design problem(s). Examples of these templates are shown in Table 1. Each of these templates has a range of properties that specify the meta-model elements and associations they refer to, critique message(s) to generate for the tool user, and model update operations that need to be performed to resolve problems.

Table 1. Constraint and Action Assertion Templates [13].

Types	Templates
Attribute Constraint	<entity> must have | may have a [unique] <attributeTerm> <attributeTerm1>must be | may be <relationalOperator> <value> | <attributeTerm2>
Relationship Constraint	[<cardinality>]<entity1> is a/an <role> of [<cardinality>]<entity2> [<cardinality>]<entity1> is associated with [<cardinality>]<entity2> <entity1> must have | may have [<cardinality>]<entity2> <entity1> is a/an <entity2>
Action Assertion	When <event> [if <condition>] then <action>

We have employed these templates for domain-specific software tool specification specifically for visual critic authoring. They have the advantage of providing critic authors and tool designers with:

- A formal language definition to define sentence patterns for rule statements;
- Structured natural language can be used to easily represent the rules and thus is a more human-centric approach to rule specification;
- Guidance for user to determine and construct and validate the rules can be provided;
- An association between rule statements and tool specification elements is supported, e.g. the template (<entity>, <attributeTerm>, <cardinality>, <role>, etc.) matches the tool specification elements properties (entity, attribute, end1Multiplicity, end2Multiplicity, association EndName, etc.);
- A critic engine can be implemented straightforwardly using Marama's event handling mechanism to instantiate and run constraint and action assertion templates;
- A range of templates can be implemented and provided for reuse via our critic definer view in Marama, but these can straightforwardly be extended by tool developers using a class reuse approach. This allows very specific rules, critiques and actions to still be implemented and be supported by our critic engine.

5. Marama Critic Authoring Example

In this section we illustrate the critic authoring process via three major components in our extended Marama metatools: the Marama critic definer, critic construction and critic feedback editors. The tool developer, or even a target end-user designer, can select the *Critic Definer* editor to construct or modify a critic after the Marama-based tool has been defined (or iteratively with its definition). Once the *Critic Definer*, Figure 3 (a), is selected, a visual critic definer editor is displayed as shown in Figure 3 (b). The visual critic definer notation has two elements: *Critic* and *CriticFeedback* and an association link to connect them. The *Critic* element allows a tool designer to specify and define critic(s), whereas the *CriticFeedback* element is to define feedback for each critic defined for the tool under development.

The *Critic* element is associated with a form-based critic construction interface designed to ease the task of authoring and defining critics. The critic construction interface, shown in Figure 4 comprises a set of reusable "critic templates" to define critics conforming to one of a number of available template forms. A range of templates based on Table 1 is provided with Marama. More experienced tool designers can extend this using a CriticClass approach similar to that of ArgoUML [2].

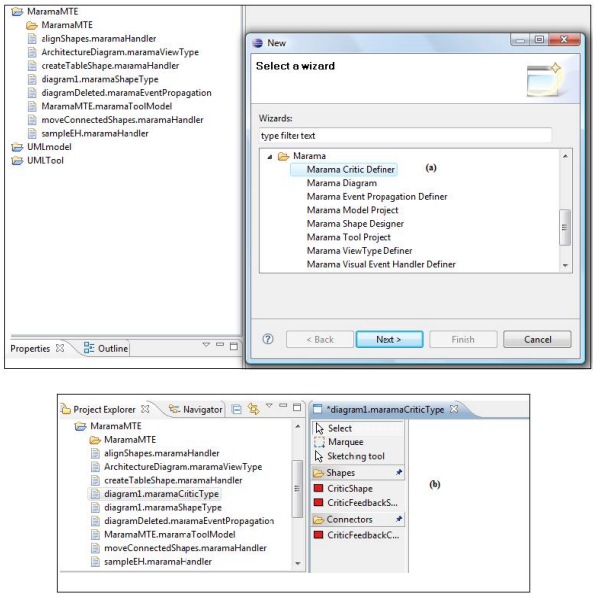

Figure 3. Marama critic definer view (a) and visual critic definer editor (b)

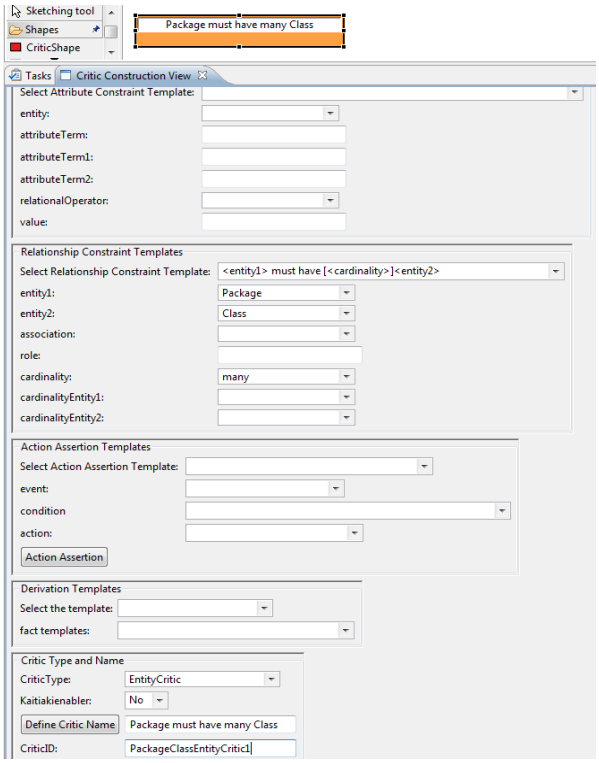

Figure 4. Critic construction view associated with *CriticShape* function

Several example templates are shown in Figure 4. To specify a critic for the selected Marama tool the critic author selects a template they wish to use and fills out the critic template using the form. The attribute constraint and relationship constraint template properties (as shown in Figure 4) allow the construction of critics that capture violations of the static semantics, while the action assertion template is to specify complex critics and behavioral semantics. In this example the critic author is specifying a constraint that Package must contains Classes: they specify items of interest (Package, Package-to-Class association, Class, and Class name property uniqueness).

The *CriticFeedback* element is associated with a critic feedback view interface that is designed to ease the task of defining critique feedback(s) for each design critic. The critic feedback view interface as shown in Figure 5 is also a form-based interface that includes properties to define critic's feedback.

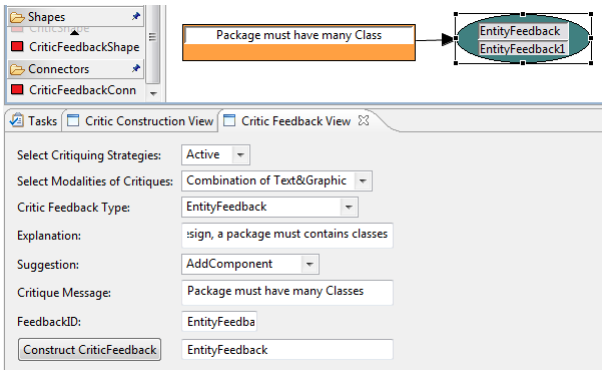

Figure 5. Critic feedback view associated with *CriticFeedbackShape* function

6. Case Study: MaramaCD Tool Critics

In this section, we present a case study, a Class Diagram (MaramaCD) tool to illustrate the use of our visual critic authoring tool. MaramaCD provides a simplified Unified Modeling Language (UML) class diagram tool which is simplified for clarity here.

Initially, a tool or end-user designer specifies the MaramaCD tool using the various meta-tool editors. A tool designer specifies a variety of entities and associations to represent the structure of a class diagram. The basic items of a class diagram are: package; class with the properties name attribute and operation; and various relationships between these items, as shown in Figure 6.

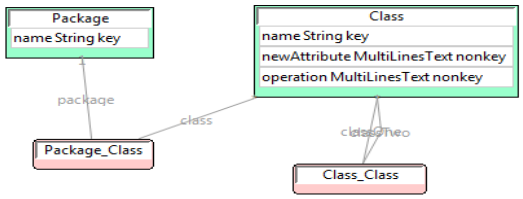

Figure 6. Metamodel for MaramaCD tool

Table 2. Examples of critics and feedbacks for MaramaCD tool

Tool element	Critic rule phrase	Critic template	Type	Feedback
Class	A class must have a unique name	<entity> must have | may have a [unique] <attributeTerm>	Attribute constraint	Remove or rename one of the components
Class (newAttribute)	newAttr may be equal or greater than zero	<attributeTerm1> must be |may be <relationalOperator> <value> | <attributeTerm2>	Attribute constraint	Add the properties
Package	Package must have many classes	<entity1> must have | may have [<cardinality>] <entity2>	Relationship constraint	Add the component
Class	When a class has too many associations then reduce the association	When <event> [if <condition>] then <action>	Action assertion	Reduce the association

Once the MaramaCD tool is defined, critics can be specified by selecting the *Critic Definer* (as shown in Figure 3). On the assumption that the tool designer has adequate knowledge of the tool domain (UML class diagrams), a set of critic rules and critic feedbacks relevant to that domain are defined via the visual critic definer. Examples of critics and feedbacks defined for the MaramaCD tool are shown in Table 2.

The critic-authoring task is completed when the tool designer is satisfied with the list of critics and feedbacks defined for the tool. These critics and feedbacks are then applied in the executing tool (i.e., at the model or Marama diagram level). Further critics and feedbacks can then be added and modified based on the tool designer preferences. Critic behavior can also be modified by the tool designer – or even the tool end-user – via the critic definer view even while the tool is in use.

A complete MaramaCD tool with critic support is now available for users to model class diagrams. For example, consider a user who wants to construct a class diagram for a Library system. If the model violates one of the critics rules defined above, then a critique related to that critic is displayed followed by a feedback that contains an explanation and a necessary action to fix the model's problem. The explanation is written by the tool designer in the critic feedback view interface on the explanation property. The fix action is based on options given in the suggestion property. The descriptions are shown in Figure 7 and Figure 8.

7. Implementation

We have implemented our visual critic authoring approach by embedding it to the Marama meta-toolset.

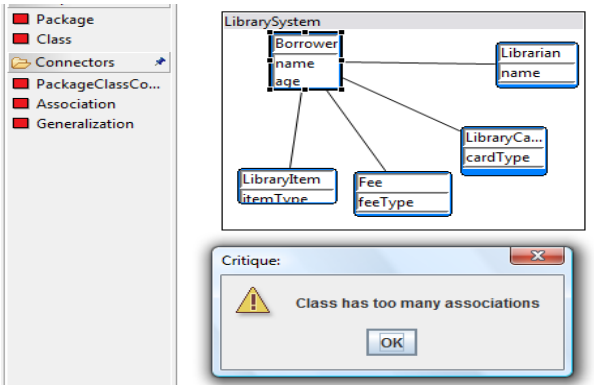

Figure 7. Critic executed at the model level

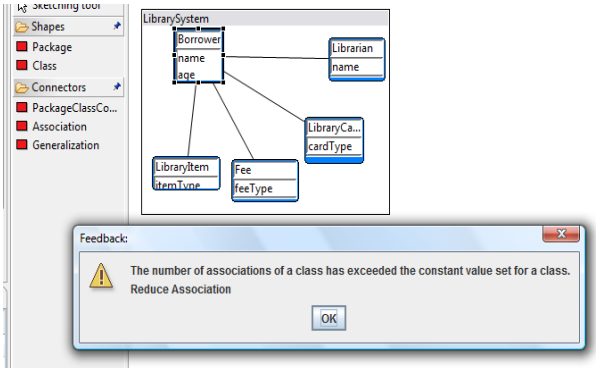

Figure 8. Critic's feedback with fix action

Three new editors: *Marama Critic Definer*, *Critic Construction*, and *Critic Feedback* are the major components that allow the end-user and tool designer to perform the critic authoring task.

The *Critic Definer* editor supports a very simple visual notation comprising only two significant visual elements for specifying critics, *Critic* and *CriticFeedback* together with a relationship connector

used to associate critics with their feedback mechanisms. The *Critic Definer* editor both augments and is itself defined using the Marama meta-toolset (as are the other Marama specification editors). We associated the *Critic* element with a form-based interface for detailed specification to simplify critic authoring. This interface, the *Critic Construction* editor, instantiates the critic authoring template mechanism providing a simple mechanism for providing property and parameter values. Critics are defined via this interface and then stored in the Marama tool specification repository. Critic feedback specification is managed in a similar way with each *CriticFeedback* element associated with a form-based interface, the *CriticFeedback* editor, to provide detailed specification of the feedback for each critic. These are also stored in the Marama tool repository. Examples are shown in Figure 9. The *Critic Construction* and *Feedback* editors use a more custom implementation than does the critic definer editor. New types of rule templates can be specified and existing rules tailored by editing textual domain-specific language specifications using an, as yet, fairly rudimentary editing environment. Ideally we would like to develop a more visual approach to these rule template specifications.

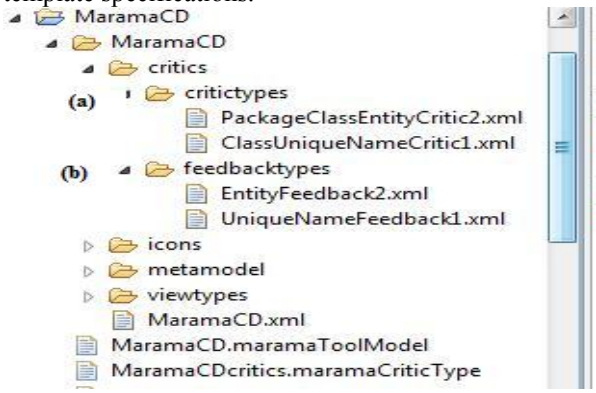

Figure 9. C*ritic* (a) and *feedback* (b) repository browser

Each critic and feedback is stored in an XML format in the tool repository. Once a critic and feedback mechanisms have been specified, a code generator template is used to construct a Java class implementing the critic. This is then instantiated into the tool when it is executed.

8. Discussion and evaluation

The implementation of *Marama Critic Definer*, *Critic Construction*, and *Critic Feedback* editors and notations contributes several benefits. These include:

- a simple way to express and define critic condition specifications based on the structured rule templates given, making it easier for novice designers to author and realize critics;
- a simple way to express and define critic feedback specifications based on structured templates also making it easier for novice designers to author and realize critics feedback;
- the process of customizing critics and their feedback is made easier through the combination of the visual critic definer editor and the two form based template editors.

However, the approach still has some limitations that we need to improve in our future work. These include:

- Constructing new critic condition and feedback templates is as yet rudimentary
- Only a limited set of critic authoring and feedback templates and actions have been implemented to date.

Following the failure of our initial non-scalable approach that looked to extend the existing meta-model editor, our new approach has been one of deconstructing the process of critic authoring into multiple design perspectives. This has meant we have ended up with several editors in place of the single combined editor we had previously. While this is contrary to some design approaches, such as REEP [3], it has meant we have been able to apply appropriate abstractions for each part of the process:

- a high-level visual overview of the critics designed for a tool;
- highly user accessible form-based rule template interfaces for detailed critic specification and customization;
- extensibility options for more experienced tool users via the rule template textual DSL.

We have used Cognitive Dimensions [10] to continuously evaluate our design. This leads to the following observations about tradeoffs made. We have focused on reducing *viscosity* (the amount of effort involved in making slight changes to the notation) and *hard mental operations*. This is achieved by simplifying critic customization and using the very accessible (good *closeness of mapping*, low *error proneness*) rule template approach. However, the resulting multiple design perspectives come at the expense of *hidden dependency* creation. To mitigate the hidden dependencies we have provided extensive navigation support to allow the impact of critic design changes to be readily understood. This allows rapid navigation to affected design views. As further steps we could extend the *terse* notation of our visual editor to automatically render meta-model elements

associated with critics in situ and provide co-highlighting of impacted elements across *juxtaposed* views.

We have attempted to provide a balance of abstractions, with a *low abstraction gradient* for end-users but with limited expressability and greater abstraction capability for more experienced developers. Based on our earlier work on critic taxonomies [1] the templates we have provided have aimed at providing the bulk of types of rule that we feel are accessible and understandable by end-user developers. Critics that require greater expressability necessarily demand better programming skills and these are supported via the template and class extension mechanisms.

Our approach supports *progressive evaluation* well. As with the other Marama meta-editors, changes made by the critic definition views in a tool take immediate effect in currently open models using that tool.

In the future, we are planning to provide a better template specification tool abstracting the current textual DSL and editor. This will allow new templates to be specified in a more visual manner, with actions realized using Marama's other visual specification tools. We plan to provide visualization of dependencies between critic and model elements via interactive hide/show. We are also in the process of developing hierarchical critics (rules that trigger other "higher level" rules) to provide a more powerful critic reasoning mechanism. Specifying critics-by-example will allow authors to interactively interact with a tool to record partially-instantiated critic templates. We also plan a larger end-user evaluation to supplement the cognitive dimensions evaluations that have been guiding our developments to date.

9. Summary

We have presented the *Marama Critic Definer* which provides support to the end-user and tool designers for critic authoring and configuration tasks. A combination of textual and visual design notation approaches is applied to ease the critic authoring task by the end-user and tool designers. We have applied our new Marama Critic Definer view to several Marama-based tools, including the MaramaMTE software architecture design tool [12] and the MaramaUCD Use Case Diagram tool. We are applying this approach with other complex Marama-based tools.

References

1. Ali, N.M., Hosking, J., Huh, J. And Grundy, J. Critic Authoring Templates for Specifying Domain-Specific Visual Language Tools, Proc. ASWEC09, Gold Coast, Australia, April 14-17 2009.
2. ArgoUML, http://argouml.tigris.org/
3. Barone, R., & Cheng, P. C.-H. (2004). Representations for problem solving: on the benefits of integrated structure. Proc 8th Intnl Conf on Information Visualisation (pp. 575-580). Los Alamitos, CA: IEEE.
4. Bergenti, F. and Poggi. A. Improving UML designs using automatic design pattern detection, Proc SEKE, 2000, pp. 336-343.
5. Costagliola, G., Denfemia, V., Ferrucci, F., and Gravino, C. "A User-centered Methodology to Generate Visual Modeling Enviroments" in *Enterprise Information Systems VI*, Springer, Netherlands, 2006.
6. Eclipse, http://www.eclipse.org/
7. ExtendedBNF, http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf
8. Fischer, G. A critic for LISP, In Proc of the 10th IJCAI (Milan, Aug.1987) pp. 177-184.
9. Florijin, G. RevJava-Design critiques and architectural conformance checking for Java software, Technical Report, White Paper, SERC 2002. http://www.bi-in-business.nl/site.nsf/0/A284738E9CD72E0EC1256E43002FE770/$file/Whitepaper_RevJava.pdf
10. Green, T.R.G. & Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 1996 (7), pp. 131-174.
11. Grundy, J.C., Hosking, J.G., Huh, J. and Li, N. Marama: an Eclipse meta-toolset for generating multi-view environments, Formal demoICSE'08, Liepzig, Germany, May 2008, ACM Press.
12. Grundy, J.C., Hosking, J.G., Li, L. and Liu, N. Performance engineering of service compositions, ICSE 2006 Workshop on Service-oriented Software Engineering, Shanghai, May 2006.
13. Loucopoulus, P., and Wan Kadir, W.M.N. "BROOD:Business rules-driven object-oriented design", *J Database Management*, 19(1), 2008, pp. 41-73.
14. Oh,Y., Gross, M.D and Do, E.Y.-L, Computer-aided critiquing systems, lessons learned and new research directions. http://code.arc.cmu.edu/lab/upload/caadria-oh.0.pdf
15. Qiu, L., and Riesbeck, C.K., An incremental model for developing educational critiquing systems: experiences with the Java critiquer, *J Interactive Learning Research*, 2008(19), pp.119-145.
16. Robbins, J.E., Redmiles, D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *IST* 2000, 42(2) pp. 71-149.
17. Robbins, J.E., Hilbert, D.M. Redmiles, D.F. Software Architecture Critics in Argo. Intelligent User Interfaces 1998, pp. 141-144.
18. Souza, C.R.B., et al. A Group Critic System for Object-Oriented Analysis and Design, Proc ASE 2000, 313-316.
19. Wan Kadir, W.M.N., and Loucopoulus, P. "Relating evolving business rules to software design", *Journal of Systems Architecture*, 50(7), Elsevier, 2004, pp.367-382.