Pounamu: a meta-tool for multi-view visual language environment construction

Nianping Zhu[1], John Grundy[1, 2] and John Hosking[1]
Department of Computer Science[1] and Department of Electrical and Computer Engineering[2],
University of Auckland, Private Bag 92019, Auckland, New Zealand
{nianping | john-g | john}@cs.auckland.ac.nz

Abstract

We describe a meta tool for specification and generation of multiple view visual tools. The tool permits rapid specification of visual notational elements, the tool information model, visual editors, the relationship between notational and model elements, and behaviour. Tools are generated on the fly and can be used for modelling immediately. Changes to the meta tool specification are immediately reflected in tool instances.

1. Introduction

Multi-view, multi-notational visual environments are popular tools in a wide variety of domains ranging from software design tools to circuit designers. Many frameworks, meta-tool environments and toolkits have been created to help support the development of such visual language environments. These include MetaEdit+ [3], Meta-MOOSE [1], GME [5], Escalante [6], IPSEN [4] and DiaGen [7]. We have had a long term interest in developing frameworks and meta-tools supporting development of such tools, including the MViews/JViews framework and JComposer/BuildByWire meta-tools [2].

However, current approaches to developing such tools suffer from several deficiencies. Tools may be easy to learn and use but support only a limited range of target visual environments, or, tools may be flexible but require considerable programming to develop. In addition, most meta-tools have an edit-compile-run cycle, requiring complex tool regeneration for minor changes. Our aim was to produce a new meta-tool, Pounamu[1], to rapidly design, prototype and evolve tools for a wide range of visual notations. We based Pounamu's design on two overarching requirements: *simplicity of use* and *simplicity of extension and modification*.

2. Tool Specification using Pounamu

Five sub-tools are used to create a Pounamu tool meta description. Figure 1 (a) shows the *shape designer* in use. A hierarchical view (left) provides access to tool specification components and instantiated models. In centre are editing windows for defining tool components and model instances. Here, a shape is defined for a generic UML class icon. This consists of Java Swing panels, with embedded sub-shapes, such as labels, editable text fields, layout managers, geometric shapes, images, borders, etc. To the right is a property editing panel supplementing the visual editing window. This allows names and formatting information to be specified and exposed for each shape component. General information is provided in a panel at the bottom. The *connector designer* allows specification of connectors, eg a UML generalisation connector in Figure 1(b). The tool permits specification of line format, end shapes, and adjacent labels or edit fields.

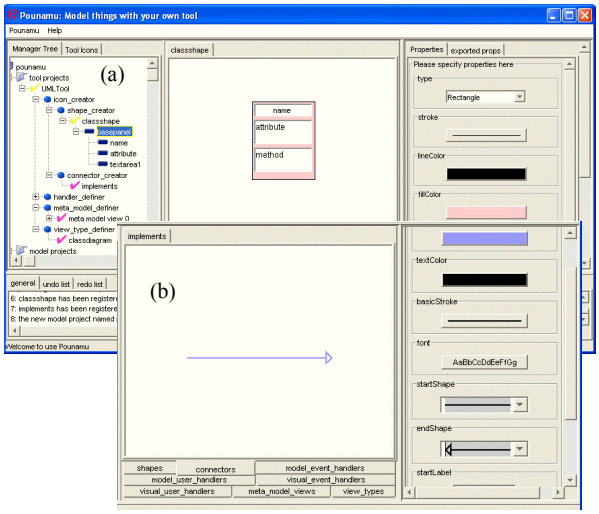

Figure 1. Pounamu in use specifying (a) a visual notation shape element and (b) a connector.

The underlying tool information model is specified using the *meta model designer*. This uses an Extended Entity Relationship model as its metaphor, chosen as it is simple and accessible to a wide range of users. The meta model in Figure 2(a) has two entities representing a UML class and object, each with properties for their name, attributes and methods, class type etc. An "instanceOf" association links class and object entities and "implements" association links classes. The meta model tool has multiple views, so meta models can be broken into manageable segments.

The *view designer*, is used to define a visual editor and its mapping to the information model.

[1] *Pounamu*: Maori word for greenstone jade, used to produce tools, and objects of beauty, or *taonga*.

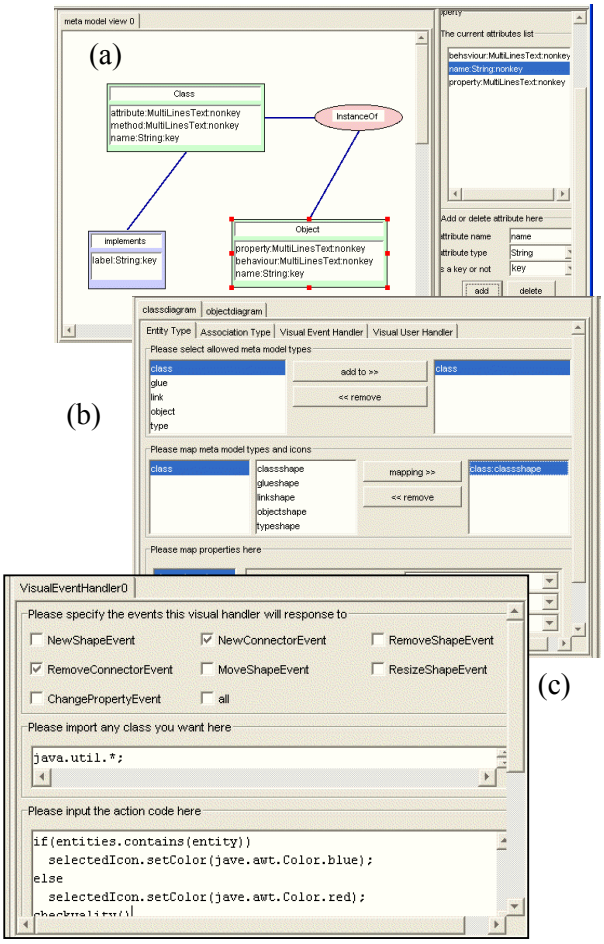

Figure 2 Examples of (a) meta-model designer (b) view designer (c) event handler designer.

Each view type consists of the allowable shape and connector types, and a mapping from each to corresponding meta model types. Figure 2(b) shows the specification of a simple UML class diagramming tool, consisting of class icon shapes, and generalisation connectors. Multiple view types can be defined mapping to a common information model.

An event handler adds complex behaviour to a tool by specifying the event type(s) that causes it to be triggered (eg shape/connector addition/modification), filtering condition e.g. property value, and response (i.e. action to take) in the form of a piece of Java code. An API provides access to the underlying tool representation. Handlers are typically used to add constraints, complex mappings, back end data import/export, code generation, and access to remote services to support tool integration and extension. Handlers are specified using the *handler designer* (Figure 2(c) and included in a tool via view and meta-model tools.

3. Modelling Tool Example Usage

To use a tool, a user opens the tool project(s) required and Pounamu dynamically initialises the tool facilities specified by the tool project(s). Generation of the tool happens automatically and immediately following specification of a view editor associated with the tool. Users can create model views using any of the specified view editors. Each view editor provides an editing environment for diagrams using the shapes and connectors it supports. Consistency between multiple views is implicitly supported via the view mapping process with no programming required to achieve this, unless very complex mappings are required that need event handlers to implement them. Figure 3 shows the simple UML class diagramming tool in use. View (1) shows a simple class diagram with two UML class shapes and an association connector. View (2) shows another class diagram, reusing the *Customer* class information.

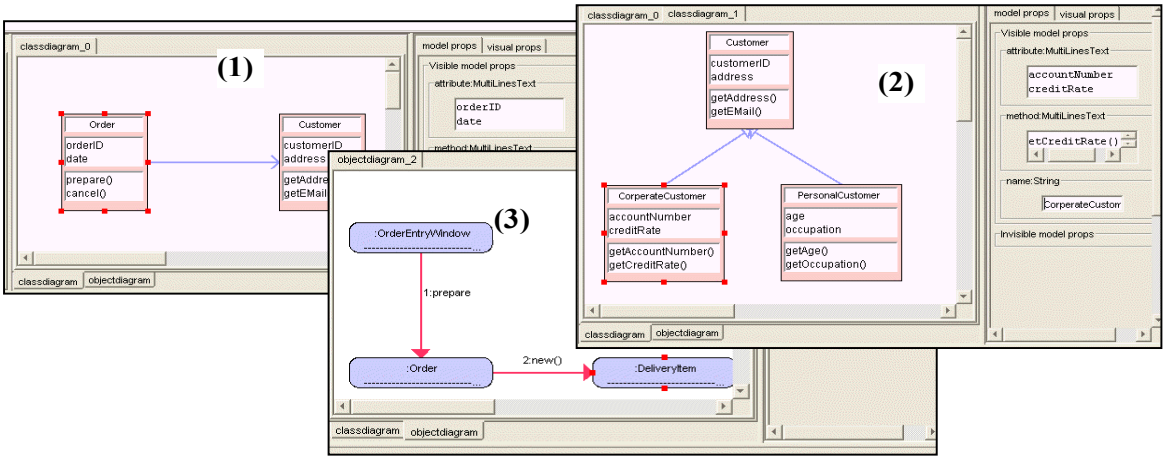

Figure 3. Example UML modelling tool usage.

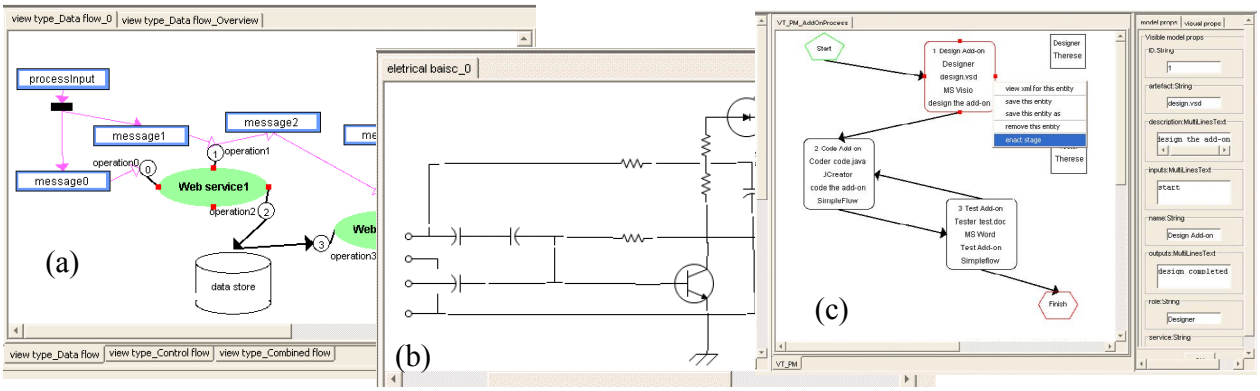

Figure 4. Pounamu exemplar tools: (a) web services integration (b) circuit designer (c) process modeller.

Changes to either view are reflected through to the other view. View (3) shows an object diagram view, with an object of class *Order*. Changes to the class name are reflected in this view and only methods defined or inherited by a class may be used in message calling. The latter is controlled by event handlers managing the consistency requirement.

4. Tool Modification and Extension

Our second requirement was simplicity of extension and modification. Users can at any time modify tool specifications. Changes are immediately reflected in models being edited using that tool. This provides powerful support for evolutionary development.

Having defined a simple tool, and experimented with its notation, additional behaviour can be added using event handlers to implement more complex constraints (eg uniqueness of class names) or add back end functionality (e.g. generate C# skeleton code from model instances). Back end support can also be added using XSLT or other XML-based transformation tools applied to the XML based tool representation or via a web services-based API.

5. Example Applications

We have evaluated Pounamu's suitability for multiple-view visual language environment development by using it to implement a wide variety of tools and evaluating the development process against our primary requirements. These include tools for design in UML supporting all Some of these include a full UML tool supporting all major view types; electrical circuit modelling, semantic modelling using Traits, web services system integration, and software process modelling integrated with a process enactment engine. Examples are shown in Figure 4.

In each case Pounamu permitted rapid development of an environment for a simple version of the supported notation, satisfying our first requirement. These tools were then iteratively expanded in a manner matching the second of our requirements. This involved, for example:

- elaboration of notations, such as expansion of the range of UML diagrams supported in the UML tool
- addition of event handlers for constraint management, particularly for visual constraints and for consistency management between elements in the information model.
- integration of backend code generation for the web services and process modelling tools, and
- use of the web services API to integrate the process modelling tool with a process enactment engine.

Acknowledgements

We acknowledge the financial support of the New Zealand Foundation for Research Science and Technology's New Economy Research Fund.

References

[1] Ferguson R, Parrington N, Dunne P, Archibald J, Thompson J, MetaMOOSE-an object-oriented framework for the construction of CASE tools, In Proc Int Symp on Constructing Soft. Eng. Tools (CoSET'99) LA, May 1999.

[2] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Visual specification of multiple view visual environments, In Proc IEEE VL'98, Halifax, Nova Scotia, Sept 1998, pp. 236-243

[3] Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in *Proceedings of CAiSE'96*, LNCS 1080, Springer-Verlag, Crete, Greece, May 1996.

[4] P. Klein, A. Schürr: Constructing SDEs with the IPSEN Meta Environment, In *Proc SEE'97*, pp. 2-10

[5] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: Composing Domain-Specific Design Environments, *Computer*, 44-51, Nov, 2001.

[6] J.D. McWhirter and G.J. Nutt, Escalante: An Environment for the Rapid Construction of Visual Language Applications, In *Proc. VL '94*, pp. 15-22, Oct. 1994.

[7] M. Minas and G. Viehstaedt, DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, *Proc. VL '95,* 203-210 Sept. 1995.