Constructing domain-specific design tools with a visual language meta-tool

Nianping Zhu[1], John Grundy[1, 2] and John Hosking[1]

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

Abstract. Collaborative, visual design tools are typically difficult to build and evolve. We describe a meta tool for specification and generation of multiple view, multiple user visual design tools. The tool permits rapid specification of visual notational elements, underlying tool information model requirements, visual editors, the relationship between notational and model elements, and behavioural components. Tools are generated on the fly and can be used for modelling immediately. Changes to the meta tool specification are immediately reflected in any tool instances.

1 Introduction

Multi-view, multi-notational visual environments are popular tools in a wide variety of domains. Examples include software design tools, circuit designers, visual programming languages, user interface design tools, and children's programming environments. Many frameworks, meta-tool environments and toolkits have been created to help support the development of such visual language environments. These include MetaEdit+ [6], Meta-MOOSE [2], Escalante [8], and DiaGen [9]. We have also had a long term interest in developing frameworks and meta tools supporting development of such tools, including JViews [3] and JComposer meta-tools [4].

However, current approaches to developing multiple-view visual language tools suffer from several deficiencies. Frameworks provide low-level yet very powerful sets of reusable facilities for building specific kinds of visual language tools or quite general-purpose applications, depending on their degree of domain specialisation. General purpose frameworks like MVC [5] and Unidraw [11] typically lack abstractions specific to multi-view, visual language environments. Special purpose frameworks like Meta-MOOSE [2], JViews [3], and Escalante [8] offer more easily reusable facilities for visual language environments, but require detailed programming knowledge and a compile/edit/run cycle, limiting their ease of use and flexibility for exploratory development. Many general-purpose toolkits that are suitable for visual language development have been produced, including Tcl/Tk [12] and Suite [1], but lack high-level abstractions for visual, multi-view environments. More targeted toolkits include DiaGen [9], JComposer [4] and PROGRES [10]. Some use a code gen-

Proceedings of the CAiSE'05 Forum - O. Belo, J. Eder, J. Falcão e Cunha, O. Pastor (Eds.)

eration approach from a specification, e.g DiaGen and JComposer. Others, such as PROGRES, are based on formalisms such as graph grammars and graph rewriting which are used for high-level syntactic and semantic specification of tools. Code generation approaches suffer from similar problems to many toolkits: edit/ compile/run cycle needed and difficulty in integrating third party solutions. Meta-tools provide an integrated environment for developing other tools. These include MetaEdit+ [6], Escalante [8], JComposer [4] and IPSEN [7]. Typically meta-tools provide good support for their target domain environments but they are often limited in their flexibility and degree of integration with other tools.

Our aim was to produce a new meta-tool, Pounamu[1], that could be used to rapidly design, prototype and evolve tools supporting a very wide range of visual notations and environments, ameliorating these deficiencies. To achieve this we based Pounamu's design on two overarching requirements: *Simplicity of use* and *Simplicity of extension and modification*.

2 Overview of Pounamu

Figure 1 shows the main components of the Pounamu meta-tool. A user of Pounamu initially specifies a meta-description of the desired tool. Specification tools allow definition of the appearance of visual language notation components ("Shape Designer"), views for graphical display and editing of information ("View Designer"), the tool's underlying information model as meta-model types ("Meta-model Designer"), and event handlers to define behavioural semantics ("Event Handler Designer").

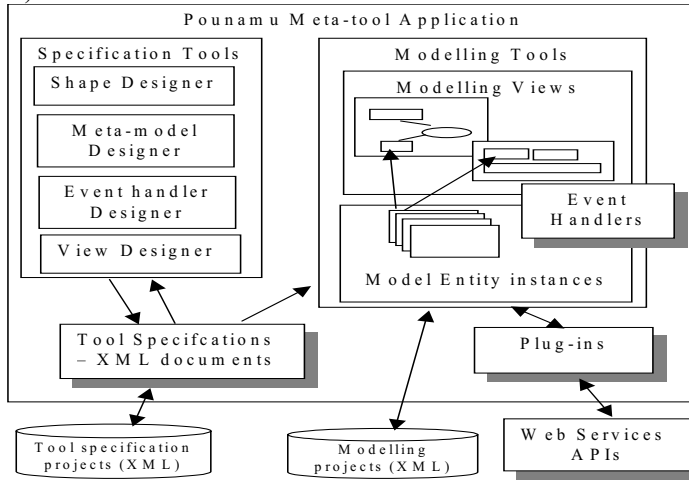

Figure 1. The Pounamu approach.

[1] *Pounamu* is the Maori word for greenstone jade, used by Maori to produce tools, such as adzes or knives, and objects of beauty, or *taonga*, such as jewellery.

Tool projects are used to group individual tool specifications. Having specified a tool or obtained someone else's tool project specification, users can create multiple project models associated with that tool. Modelling tools allow users to create modelling projects, modelling views and edit view shapes, updating model entities. To support our ease of use requirement, the shape, view and meta-model designers use high-level visual programming tools with relatively simple appearance and semantics. To ensure flexibility and openness of the tools, the event handler designer allows tool designers to choose predefined event handlers from a library or to write and dynamically add new ones as Java plug-in components. Pounamu uses an XML representation of all tool specification and model data, which can be stored in files, a database or a remote version control tool. Pounamu also provides a full web services-based API used to integrate the tool with other tools, or to remotely drive the tool.

3 Tool Specification and Usage with Pounamu

Figure 2 (1) shows an example of the Pounamu shape designer in use. On the left a hierarchical view provides access to tool specification components and models instantiated for that tool. In the centre are visual editing windows for defining tool specification components and model instances. Here, a shape is defined representing a generic UML class icon. To the right is a property editing panel supplementing the visual editing window. General information is provided in a panel at the bottom.

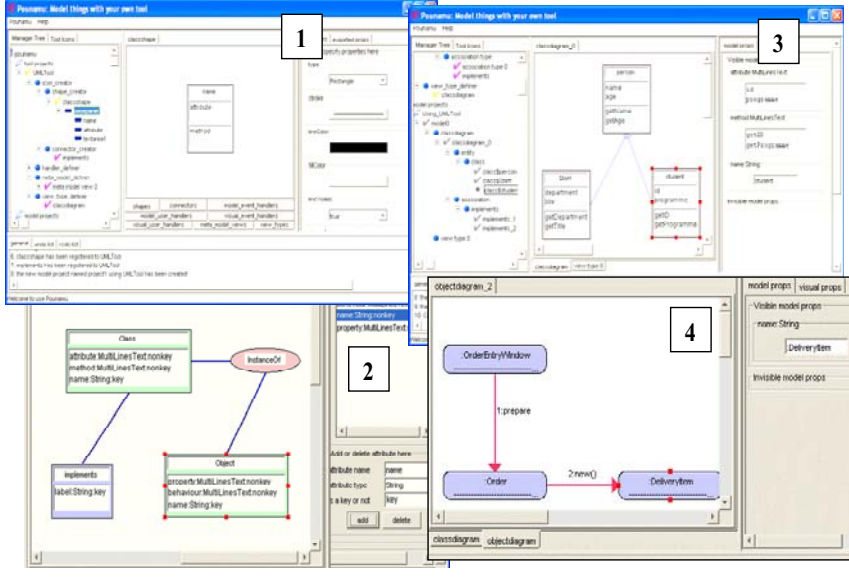

Figure 2. Pounamu in use: (1-2) specifying a tool; (3-4) using a tool.

The underlying tool information model is specified using the meta model designer, as in Figure 2 (2). This uses an Extended Entity Relationship (EER) model as its

representational metaphor with extensions for specifying complex property data types and calculated fields. In this example a meta-model contains entities representing a UML class and UML object (squares), with properties for their names, attributes and methods. An association (instanceOf) links class and object entities and another association (implements) links classes. The meta model tool supports multiple views of the meta model, allowing complex meta models to be presented in manageable segments.

Other Pounamu specification tools include the connector designer, view type designer and an event handler designer. The view designer is used to define a visual editor and its mapping to the underlying information model. Each view type consists of the shape and connector types that are allowed in that view type, together with a mapping from each such element to corresponding meta model element types. Menus and property sheets for the view editor and view shapes can also be customised using this tool. Event handlers are used to add complex behaviour to a tool via an Event-Condition-Action (ECA) model. Each handler specifies the event type(s) that causes it to be triggered (eg shape/connector addition/modification, information model element change, or user action), any event filtering condition that needs to be fulfilled e.g. property value, and the response to that event in the form of a piece of Java code.

Figure 2 (3) shows the simple UML class diagramming tool in use. View (3) shows a simple class diagram where the user has created the diagram view from the available view types, added three UML class shapes and two association connectors, and set various properties for these, including their location and size. View (4) shows a simplified object diagram view, including an object of class *Order*. Changes to the class name are automatically reflected in this view and only methods defined or inherited by a class may be used in the message calling.

4 Tool Modification and Extension

Users can at any time modify Pounamu tool specifications. Changes made are immediately reflected in models being edited using that tool, creating a live environment. This provides powerful support for rapid prototyping and evolutionary tool development. Changes to the specification may result in information creation or loss in the open or saved modelling projects e.g. on addition or deletion of new properties or types. Reuse is supported by allowing shapes, connectors, meta model elements, and event handlers to be imported from other tools or libraries. Multiple tool specification projects may be open when modelling, with specification of parts of the modelling tool coming from different tool specification projects.

Having defined a simple tool additional behaviour can be added using event handlers to implement more complex constraints. Examples include type checking (e.g. UML associations must be between classes); constraints (e.g. UML class attributes must have unique names for the same class); layout constraints and behaviour (e.g. auto-layout of a UML sequence diagram view when edited); more complex mappings (e.g. changes to class shape method names automatically modifiying method entity properties in the modelling tool information model); or add back end functionality

(e.g. generate C# skeleton code from model instances). Adding or modifying a handler results in "on the fly" compilation and incorporation in any executing tools.

Back end support e.g. for code generation can be implemented by event handlers. In addition, as all tool and model components are represented in XML format, it is straightforward to add back end support using XSLT or other XML-based transformation tools. This approach can allow back ends to be developed independently of the editing environment. An additional approach for back end support is via a web services-based API. This exposes Pounamu modelling commands, menu extensions, etc, allowing tight and dynamic integration of third party and other Pounamu tools.

5 Web, Mobile and Groupware Support

Pounamu-implemented visual design tools may need to be accessed in a variety of deployment scenarios and by multiple users. We have developed a set of plug-in components that utilise the web services API of Pounamu to extend any Pounamu-specified tool with web-based diagramming using either GIF or SVG images, mobile PDA and phone displays, collaborative editing of diagrams, asynchronous version control and merging support for diagrams, and CVS repository management of versions. An example of the web-based, thin-client editing interface for Pounamu tools being used is shown in Figure 3 (1). This allows a group of users to interact with Pounamu views via a web browser and standard web software infrastructure. SVG image diagrams support browser-side drag and drop of diagram component using scripting. An alternative set of software components using Nokia's MUPE framework allow users to view and edit diagrams on a wireless PDA or mobile phone. This uses a similar approach, where the MUPE server components communicate with Pounamu via its web services API and generate special MUPE XML mark-up for the view user interfaces. Figure 3 (2) shows an example of a project management Gantt chart diagram being browsed and manipulated on a mobile phone.

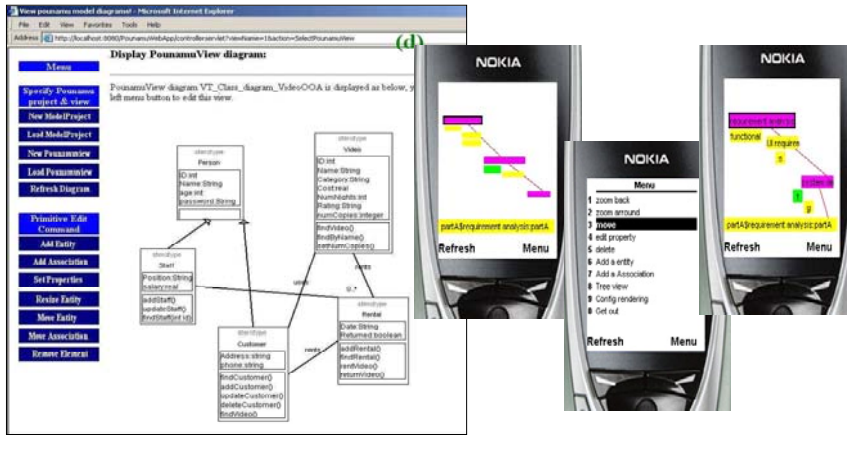

Figure 3. Thin client, web-based editing (1) and mobile editing (2) for Pounamu.

6 Summary

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 a full UML tool supporting all major view types; electrical circuit modeling, semantic modelling using Traits, web services system design using Tool Abstraction, and software process modeling, the latter integrated with a process enactment engine. 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. Current work is focusing on a visual event handler specification tool, extending the meta-model with calculated property specification support, and extending the shape definer and editing plug-ins.

7 References

[1] Dewan, P. and Choudhary, R. 1991. Flexible user interface coupling in collaborative systems, *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.

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

[3] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *J. Information and Software Technology*, Vol. 42, No. 2, pp. 117-128.

[4] 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.

[5] G.E. Krasner and S.T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal Object-Oriented Programming*, vol. 1, no. 3, pp. 26-49, Aug. 1988.

[6] 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.

[7] P. Klein, A. Schürr: Constructing SDEs with the IPSEN Meta Environment , in *Proc. 8th Conf. on Software Engineering Environments*, Cottbus, Germany, April 1997, pp. 2-10

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

[9] 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.

[10] J. Rekers and A. Schuerr, Defining and Parsing Visual Languages with Layered Graph Grammars, JVLC, vol. 8, no. 1, pp. 27-55, 1997.

[11] Vlissides, J.M. and Linton, M., Unidraw: A framework for building domain-specific graphical editors, in *Proc. UIST*'89, ACM Press, pp. 158-167.

[12] Welch, B. and Jones, K. *Practical Programming in Tcl and Tk*, Prentice-Hall, 2003.