Journal of Systems and Software 80 (2007) 1390–1407

Pounamu: a meta-tool for exploratory domain-specific visual language tool development

Nianping Zhu, John Grundy, John Hosking, Na Liu, Shuping Cao and Akhil Mehra
Department of Computer Science and Department of Electrical and Computer Engineering
University of Auckland, Private Bag 92019, Auckland, New Zealand
{ nianping, john-g, john, karen }@cs.auckland.ac.nz

Abstract

Domain-specific visual language tools have become important in many domains of software engineering and end user development. However building such tools is very challenging with a need for multiple views of information and multi-user support, the ability for users to change tool diagram and meta-model specifications while in use, and a need for an open architecture for tool integration. We describe Pounamu, a meta-tool for realising such visual design environments. We describe the motivation for Pounamu, its architecture and implementation and illustrate examples of domain-specific visual language tools that we have developed with Pounamu.

Keywords: meta-tools, meta-CASE, domain-specific languages, visual design environments

Introduction

Multi-view, multi-notational Domain-Specific Visual Language (DSVL) environments have become important and popular tools in a wide variety of domains. Examples include software design tools [20], circuit designers, visual programming languages [5], user interface design tools [38], and children's programming environments [7]. Due to the challenge of implementing such tools many frameworks, meta-tool environments and toolkits have been created to help support their development. These include MetaEdit+ [23], Meta-MOOSE [13], Escalante [34], DiaGen [37], GMF [11] and DSL Tools [16], [36].

Current approaches to developing such multiple-view visual language tools suffer from a range of deficiencies. Tools may be easy to learn and use but provide support for only a limited range of target visual environments. Alternatively, the tools may target a wide range of environments but require considerable programming ability to develop even simple environments, providing high barriers to use. Many current frameworks and meta-tools have an edit-compile-run cycle, requiring complex tool regeneration each time a minor notation change is made.

Our experiences in developing domain-specific visual languages for both research and industrial application motivated us to investigate meta-tools able to support exploratory development of such languages while mitigating the deficiencies noted above. Our new meta-tool, Pounamu[1], aims to support users to rapidly design, prototype and evolve tools supporting a very wide range of visual notations and environments. To achieve this we based Pounamu's design on two overarching requirements:

- *Simplicity of use*. It should be very easy to express the design of a visual notation, and to generate an environment for modelling using the notation.
- *Simplicity of extension and modification*. It should be possible to rapidly evolve proof of concept tools by modification of the notation, addition of back end processing, integration with other tools, and behavioural extensions (such as complex constraints).

In this paper we first motivate our work with an example Pounamu-generated visual language application and survey related work. We then overview our Pounamu toolset, describing its visual tool specification

[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.

and modelling support. We discuss Pounamu's architectural support for tool evolution, multi-user support and a variety of user interface platforms, including web-based diagramming and PDA support. Following this, we describe Pounamu's design and implementation and illustrate the versatility of the tool using several example applications we have developed. We then evaluate the utility of the tool and conclude with a summary of the contributions of this research and overview of future work directions.

Motivation

Consider the development of an environment to support software process modelling, project management and sketching out of a software design. Examples of diagrams from such a tool are illustrated in Figure 1. Such a tool would provide users with visual tools to model their work process flow (1), describe project scheduling using Gantt charts (2), and software designs using a selection of UML diagrams (3). A team of developers would want multi-user support from such an environment including the ability to version diagrams and at times to collaboratively edit diagrams remotely. For some kinds of diagram it would be useful to provide web-browser or PDA based access e.g. to support on-the-go access to, for example, project management diagrams and reports, as in Figure 1 (2). Models of software designs would need to be able to be exported to other tools e.g. 3rd party UML CASE tools, and code generated from some models. Designing and implementing such an integrated and collaborative toolset is clearly complex; ideally we want this process to be simplified by leveraging meta tool capabilities.

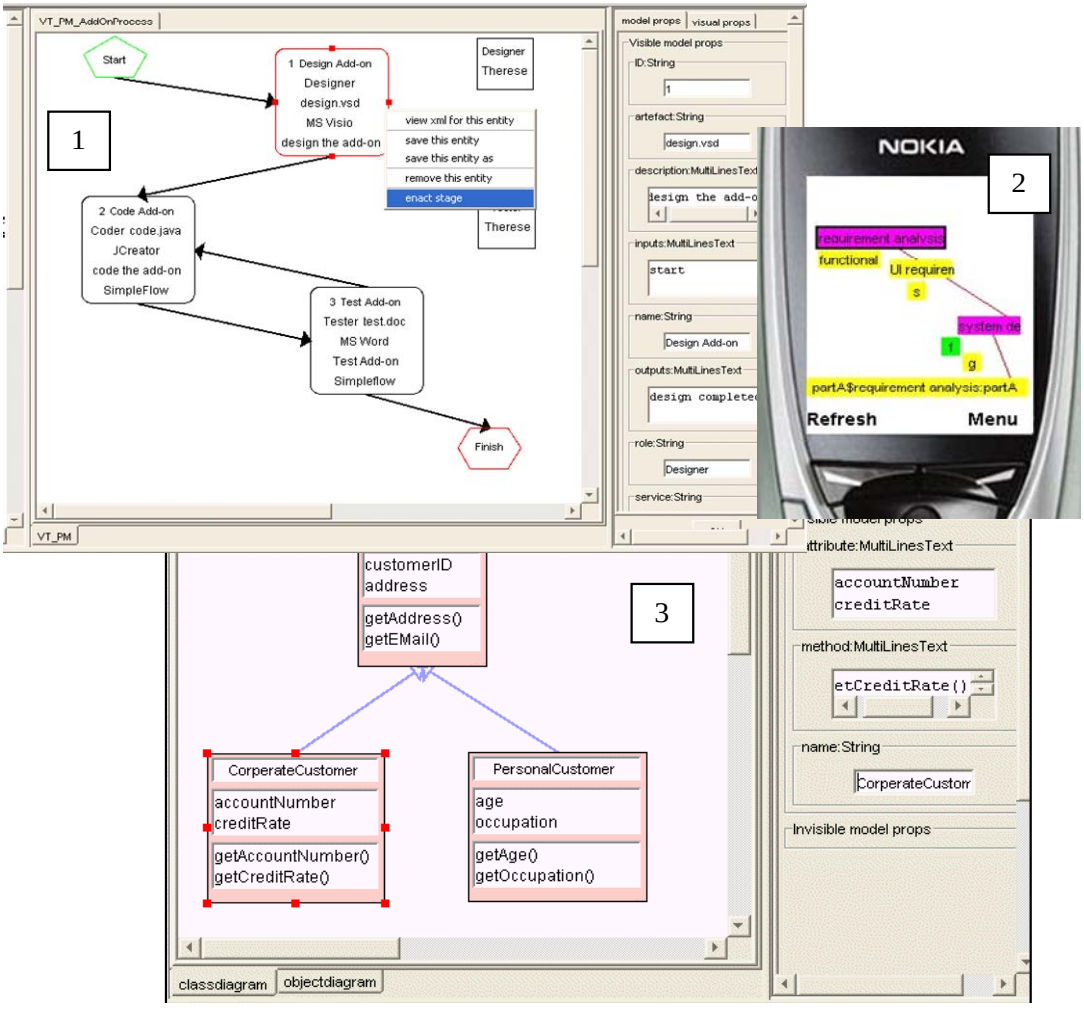

Figure 1. Some examples of DSVL tool diagrams.

While some DSVL tools, such as the ones in Figure 1, may be developed by software engineers, ideally we want to support development and modification of DSVL tools by end-users, so they can make new standalone tools, add new tools to an existing toolset or tailor existing tools to suit their needs. For example, a business process modelling tool might be prototyped by business analysts, a circuit design tool by electrical engineers or a statistical survey design tool by statisticians using their domain knowledge to develop visual notations and tools of relevance to their domain.

Together with the overarching requirements discussed earlier, key requirements for a meta-tool to construct such a range of integrated domain-specific visual language tools include:

- Visual meta-tools to specify DSVL meta-model entities and associations. These abstractions will form the canonical model of information in the target DSVL too.
- Visual meta-tools to specify simple and complex shapes and connectors. These form the building blocks for the diagrams that make up views of the model data structures. When shapes and connectors in a view are modified, the underlying model data structures need to be changed and other dependent views on these structures updated.
- Meta-tools to specify constraint handling for both view editing and model manipulation. Such constraints might include: keeping certain shapes beside or within others as diagrams are edited; computing values based on changes to other values e.g. class inherited properties from other classes in a class diagramming tool; and validation constraints over views and models e.g. all properties of a class must have a unique name.
- Generated tools are able to exchange view and model information structures with other tools e.g. export a model to another design tool, or generate code for a programming environment. The meta-tools should be able to generate tools with an open architecture and should ideally support specification of such import, export and code generation features.
- Generated tools are able to save and load models and views and to share these saved representations among multiple DSVL tools, ideally via a version control server. This supports asynchronous work by multiple DSVL tool users.
- Multiple users are able to collaboratively edit generated DSVL tool views using synchronous groupware support, including group awareness support.
- Generated DSVL tool diagrams are able to be accessed and edited across a variety of platforms, including web browsers, PDAs and mobile phones.

Related Work

Three main approaches exist for the development of the type of visual, multiple view and multi-user environment discussed in the previous section: the use of reusable class frameworks; visual language toolkits; and diagramming or CASE meta-tools.

General purpose graphical frameworks provide low-level yet powerful sets of reusable facilities for building diagramming tools or applications. These include MVC [26], Unidraw [44], COAST [42], HotDoc [4] and GEF [10]. While powerful they typically lack abstractions specific to multi-view, visual language environments, so construction of tools is time-consuming. For example, supporting multiple views of a shared model in GEF requires significant programming effort. Special purpose frameworks for building multiple user, multiple view diagramming tools include Meta-MOOSE [13], JViews [17], and Escalante [34]. These offer more easily reused facilities for visual language-based environments, but still require detailed programming knowledge and a compile/edit/run cycle, limiting their ease of use and flexibility for exploratory development.

Many general-purpose, rapid development user interface toolkits have been developed to reduce the edit/compile/run cycle. Many, including Tcl/Tk [45], Suite [8], and Amulet [38], are suitable for visual language-based tool development. They combine rapid application development tools and programming extensions. However, as they lack high-level abstractions for visual, multi-view environments and tool integration, more targeted toolkits have been produced to make such development easier. These include Vampire [32], DiaGen [37], VisPro [47], JComposer [17], and PROGRES [41]. Some of these use code generation from a specification model, e.g DiaGen and JComposer. Others, such as PROGRES and VisPro, use formalisms such as graph grammars and graph rewriting for high-level syntactic and semantic

specification of visual language tools. Code generation approaches suffer from similar problems to many toolkits: an edit/compile/run cycle is needed and many tools present difficulties when integrating third party solutions. Formalism-based visual language toolkits may limit the range of visual languages supported and are often difficult to extend in unplanned ways e.g. code generation or collaborative editing.

Meta-tools provide an integrated environment for developing other tools. These include KOGGE [9], MetaEdit+ [23], MOOT [40], GME [27], MetaEnv [1] ,IPSEN [25], GMF [11], Tiger [12], and MS DSL Tools [16], [36]. Usually they aim for a degree of round-trip engineering of the target tools. Typically they provide good support for their target domain environments, but are often limited in their flexibility and degree of integration with other tools [45]. These integration problems typically occur both at presentation (interface) and data/control levels. Many visual design tools and meta-tools have been enhanced by collaborative work facilities. These range from user interface coupling techniques [8], collaborative editing and authoring [35], and flexible, component-based architectures for collaboration [42][46][35][8]. Most of these approaches provide fixed, closed groupware functionality, however, and we desired more flexible control over target visual design tool collaborative work facilities. In addition, we wanted these capabilities to support integration with existing tools.

A number of researchers have identified the need to support thin-client access to visual design tools and various thin-client software engineering tools have been developed to exploit web-based delivery of information. Some examples include MILOS [31], a web-based process management tool, BSCW [3], a shared workspace system, Web-CASRE [30] which provides software reliability management support, web-based tool integration, and CHIME [21], which provides a hypermedia environment for software engineering. Most of these tools provide conventional, form-based web interfaces and lack web-based diagramming tools. Some recent efforts at building web-based diagramming tools have included Seek [22], a UML sequence diagramming tool, NutCASE [14], a UML class diagramming tool, and Cliki [33], a thin-client meta-diagramming tool. All of these have used custom approaches to realise the thin-client diagramming tool. They also provide limited tool tailorability by end users and limited integration support with other software tools.

In summary, the majority of the approaches we have outlined require detailed programming and class framework knowledge or understanding of complex information models (eg graph grammars). Few of these environment development tools support round trip engineering and live, evolutionary development. Regeneration of code can be a large problem when integrating backend code and most provide very limited, if any, collaborative work facilities. Almost no meta-tools or tool frameworks we are aware of support thin-client diagramming.

Overview of Pounamu

Figure 2 shows the main components of Pounamu. Users initially specify a meta-description of the desired tool via a set of visual specification tools. These define:

- The appearance of visual language notation components, via the “Shape Designer”, which has shape and connector variants;
- Views for graphical display and editing of information, via the “View Designer”;
- The tool’s underlying information model as meta-model types, via the “Meta-model Designer”; and
- Event handlers to define behaviour semantics, via the “Event Handler Designer”, which has visual and textual variants.

Tool projects are used to group individual tool specifications.

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. Event handlers can be used to add:

- view editing behaviour e.g. “if shape X is moved, move shape Y the same amount”;
- view and model constraints e.g. “all instances of entity Z must have a unique Name property”;
- user-defined events e.g. “check model is consistent when user clicks button”;
- event-driven extensions e.g. “generate C# code from the design model instance information”; and

- environment extension plug-ins e.g. “initialise the collaboration plug-in to support synchronous editing of a shared Pounamu diagram by multiple users”.

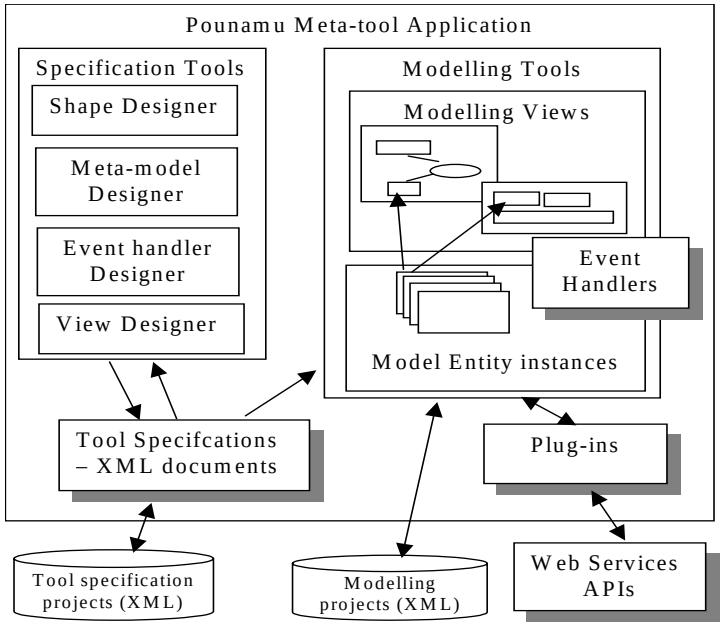

Figure 2. The Pounamu approach.

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. 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 provides a full web services-based API which can be used to integrate the tool with other tools, or to remotely drive the tool.

Tool Specification

Figure 3 (a) 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.

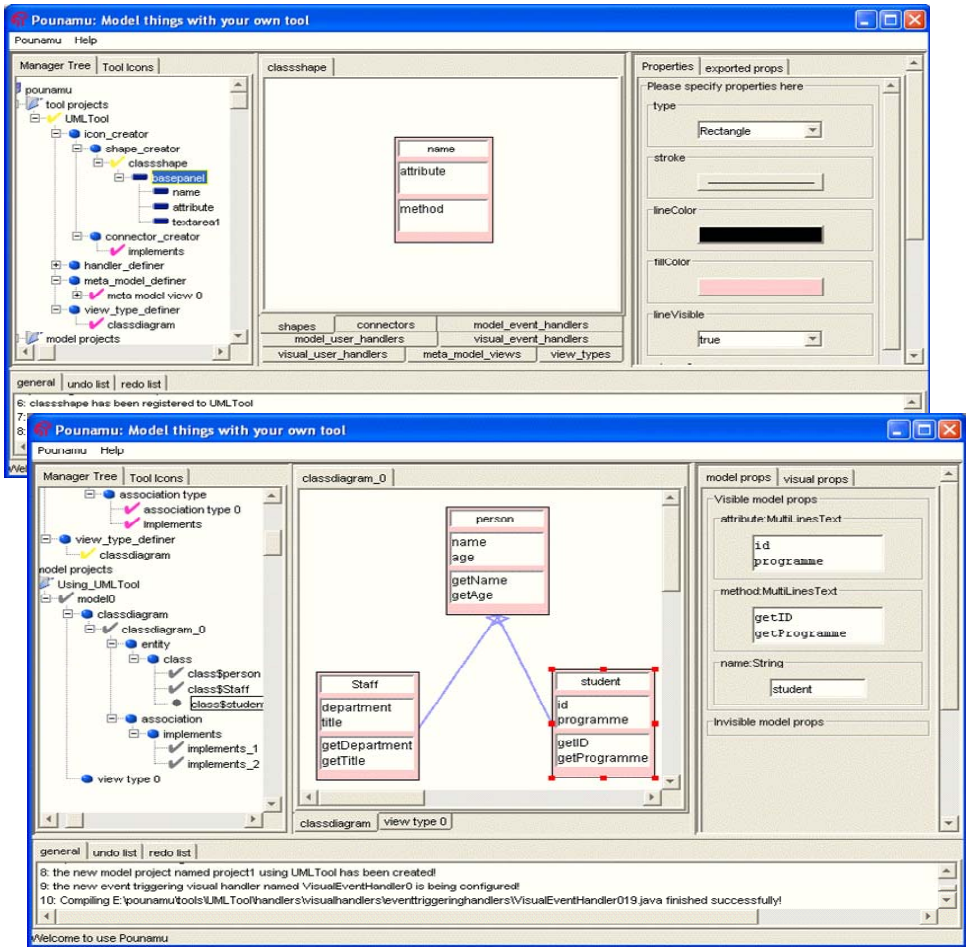

Figure 3. Pounamu in use: (a) specification of a visual notation shape element and (b) modelling using this shape in a UML class diagram tool.

Figure 3 (b) shows a UML class diagramming tool, which uses the shape icon defined in Figure 3 (a) to model a *person* class, and two subclasses *student* and *staff*. The same shape specification could be reused for other modelling tools associated with the same (eg a class element in a package diagram) or different metamodel elements.

The *shape designer* allows visual elements (generalised icons) to be defined. These consist of Java Swing panels, with embedded sub-shapes, such as labels, single or multi-line editable text fields (with formatting), layout managers, geometric shapes, images, borders, etc. For example, the icon in Figure 3 (a) consists of a bordered, filled rectangular panel, with three sub-shapes, a single line textfield for the name, and two multi-line textfields for the attribute and operation parts of the class icon. The property sheet pane (right) allows names and formatting information to be specified for each shape component. Fields that are to be exposed to the underlying information model are also specified using a property sheet tab. Form-based interfaces can also be defined by using a single shape specification defining the whole form.

The *connector designer* allows specification of inter-shape connectors, such as the UML generalisation connector shown in Figure 4. The tool permits specification of line format, end shapes, and labels or edit fields associated with the connector's ends or centre.

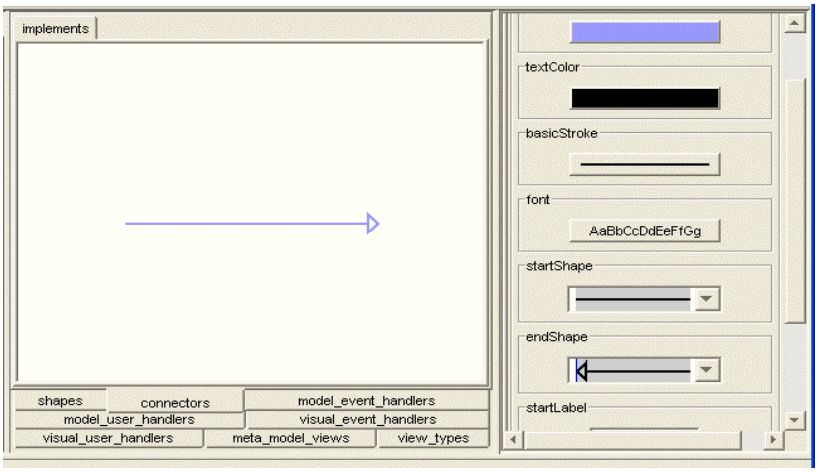

Figure 4. Example of the Connector designer.

The underlying tool information model is specified using the *meta model designer*, shown in Figure 5. This uses an Extended Entity Relationship (EER) model as its representational metaphor. This was chosen because the representation is simple and hence accessible to a wide range of users. For example, the meta model in Figure 5 contains two entities representing a UML class and UML object, each with properties for their names attributes and methods, class type etc. An "instanceOf" association links class and object entities and an "implements" association links classes. The meta model tool supports multiple views of the meta model, allowing complex meta models to be presented in manageable segments.

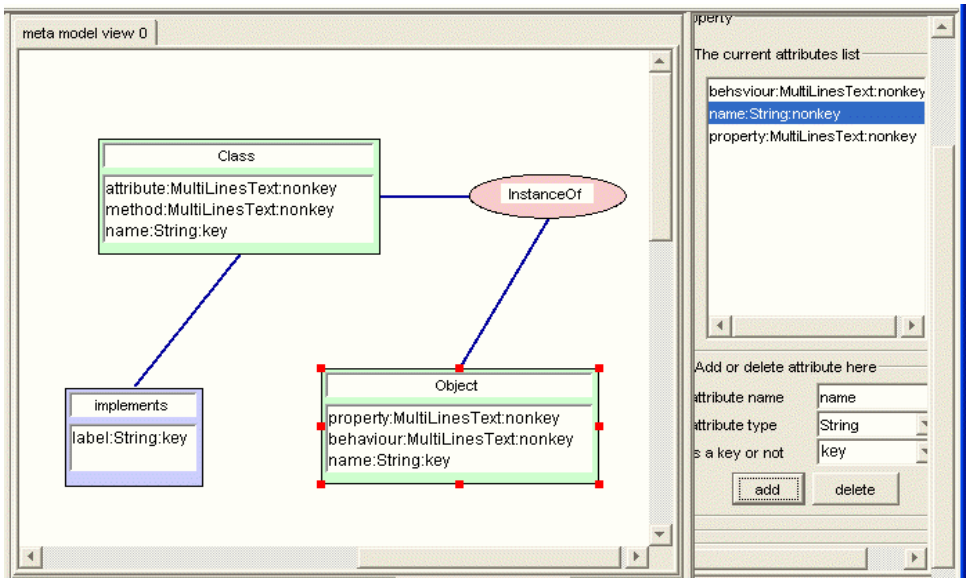

Figure 5. Example of the meta-model designer.

The *view designer*, shown in Figure 6, 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 metamodel element types. Menus and property sheets for the view editor and view shapes can also be customised using this tool. For example, Figure 6 shows the specification of a simple UML class diagramming tool, consisting of UML class icon shapes, and generalisation connectors. Figure 6 shows that the *classshape* icon maps to the *class* meta-model entity type, and their selected properties map as shown. Mappings supported in this tool are limited to simple 1-1 mappings of elements (single or multi-valued) between view instance and information model instance. More complex mappings can be specified using event handlers as described below.

Multiple view types can be defined, each mapping to a common information model. For example, other view types for sequence diagrams or package diagrams can be defined for the simple UML tool.

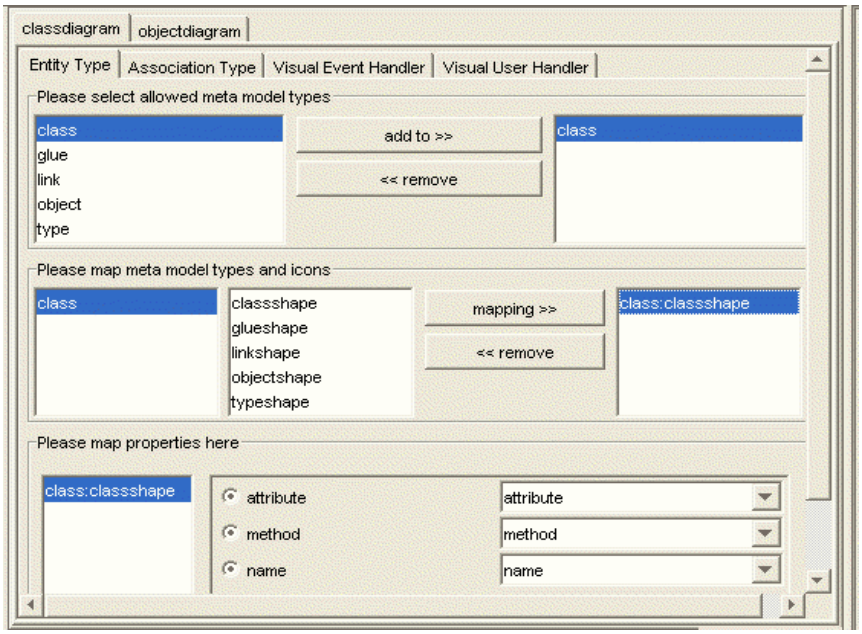

Figure 6. Example of the view designer.

In Pounamu *event handlers* add complex behaviour to a tool using an Event-Condition-Action (ECA) model [28]. Handlers are typically used to add constraints, complex mappings, back end data export or import e.g. code generation, and access to remote services to support tool integration and extension. Each handler specifies:

- the event type(s) that causes it to be triggered, e.g. shape/connector addition/modification, information model element change, or user action;
- any event filtering condition that needs to be fulfilled e.g. property value of shape or entity; and
- the response to that event, i.e. action to take, as a set of state changing operations.

A visual *event handler definer* provides a high-level visual specification based on dataflow for building both simple and complex event handling functionality for Pounamu tools from a set or reusable building blocks. A simple example event handler specification is shown in Figure 7 (a).

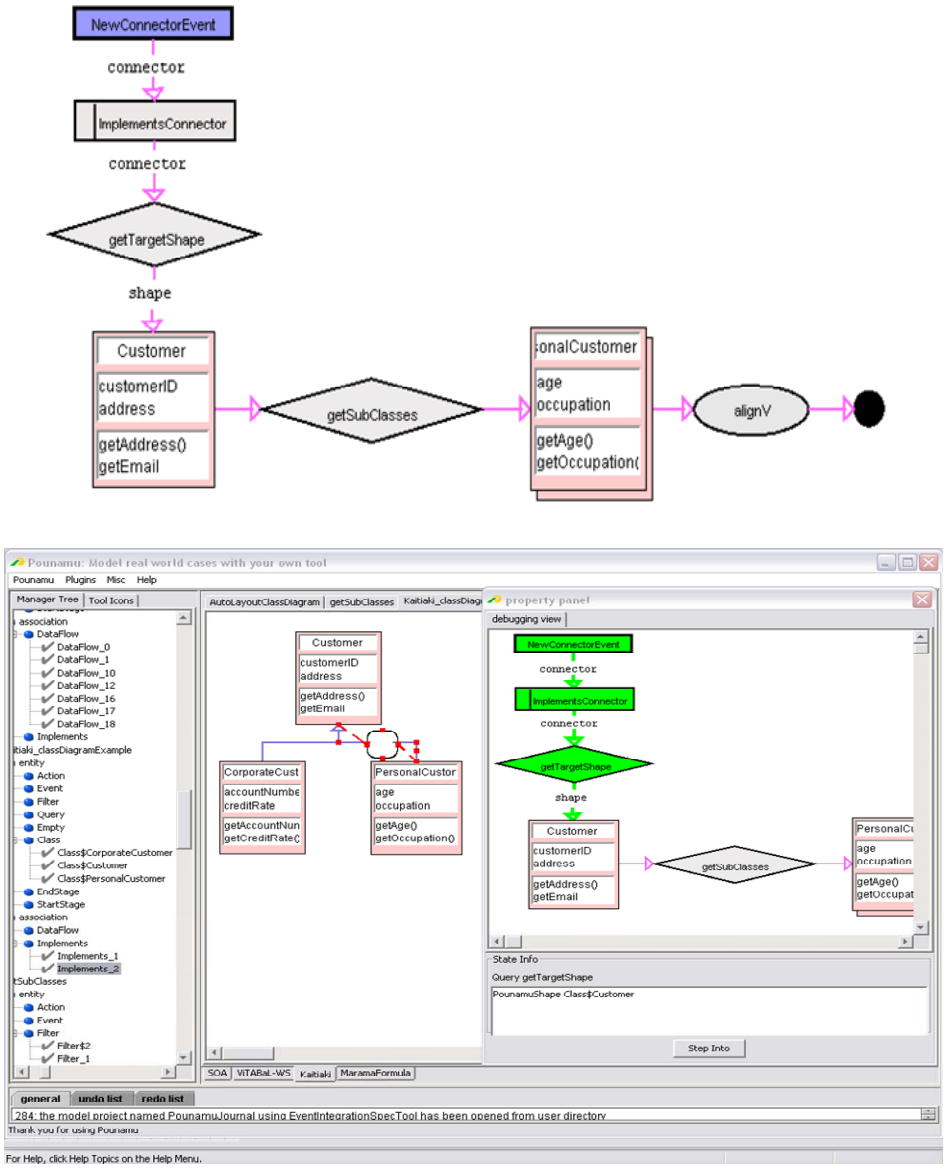

Figure 7. (a) Simple visual event handler specification; and (b) event handler debugging.

In this example an event handler for the UML tool provides automatic layout of subclasses of a parent class for a UML class diagram. This event handler is defined to respond to a built-in Pounamu *NewConnectorEvent.* The modelling constructs contained in this event handler specification include an event, *NewConnectorEvent*, which has the *connector* that has been created as data flowing from it; a filter, *ImplementsConnector* that allows only *Implements* connectors to flow through it; a primitive query *getTargetShape* to locate the end shape of the connector and flow it on to a composed query *getSubClasses* which locates the classes that implement the parent class. The data flows through data propagation links to provide input to connected entities. Type compatibility of the data sender and consumer for each dataflow is statically checked. Also incorporated in the simple event handler example are two end-user target tool icons, one on the data flow between the *getTargetShape* query and the getSubClasses query and the other on the dataflow between the *getSubClasses* query and into the *alignV* action. These are annotations that provide a visual indication of the type of data propagating along the links (a class icon, and collection of class icons respectively). We have also developed a visual debug viewer which dynamically annotates an event handler specification view during execution as shown in Figure 7 (b). This includes the visualization of event handler element invocation (by flashing the corresponding graph node) and visualization of data

propagation (by highlighting the dataflow path). The traditional “debug and step into” metaphor is used with step-by-step visualization controlled by menu commands.

For more complex event handlers and to allow expert users to add new condition and action building blocks to Pounamu’s event handler library, event hander code can also be developed using Java. A comprehensive API provides access to the underlying Pounamu modelling tool representation, permitting complex querying and manipulation of tool data. Event handlers may be parameterised and added to one or more tool specification project libraries and then reused by either the visual event handler or code-based event handler specification tools. Code-based handlers are specified using the *handler designer* and included in a tool via the view and meta-model designer tools. A simple example of a code-based event handler being developed is shown in Figure 8. Event handler code is compiled on the fly as the tool is specified or when a tool project is opened.

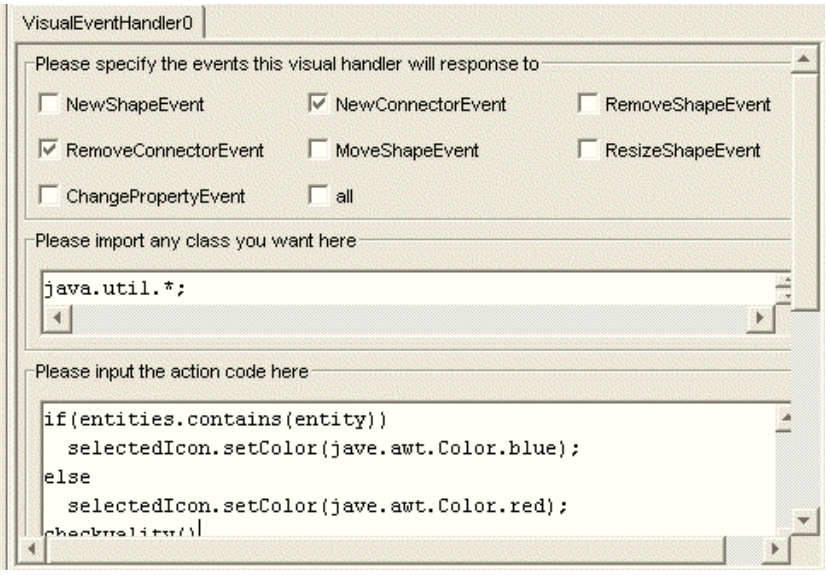

Figure 8. Example of the event handler designer.

Tool Usage

Pounamu automatically and incrementally implements tools as they are specified using the Pounamu metatools. This means tools may be tested and evaluated incrementally as they are being developed, avoiding the compile cycle issues noted earlier and creating a live environment. Generation of the tool happens automatically and immediately following specification of any view editor associated with the tool or when a saved tool project is opened. This provides powerful support for rapid prototyping and evolutionary tool development. Changes to a tool specification may, of course, result in information creation or loss in the open or saved modelling projects e.g. when adding or deleting properties or types. Users can create model views using any of the specified view editors. Reuse is supported by allowing shapes, connectors, meta model elements, and event handlers to be easily 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, supporting layered tool development

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 complex mappings are required that need event handlers to implement them.

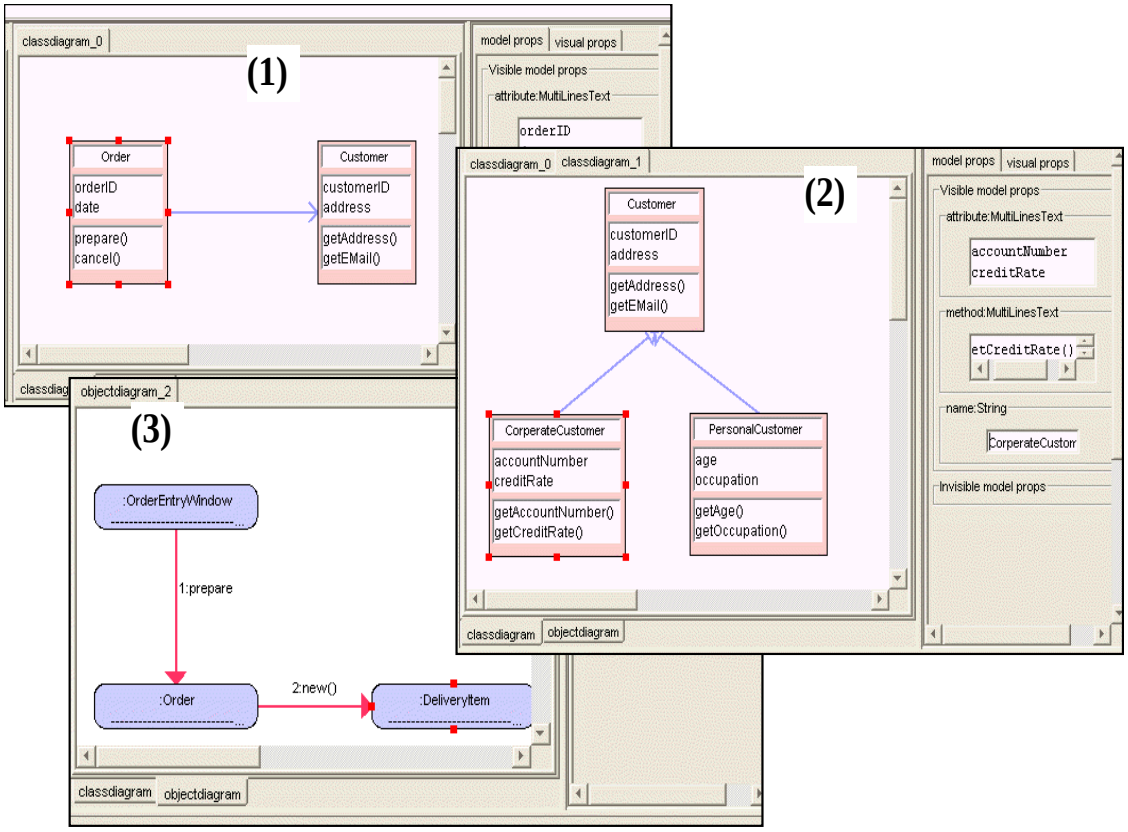

Figure 9. Example modelling tool usage.

Figure 9 shows the simple UML class diagramming tool in use. View (1) shows a simple class diagram. The user has created a class diagram view from the available view types, added two UML class shapes and an association connector, and set various properties for these, including their location and size. View (2) shows another class diagram included in the same project model, reusing the *Customer* class information. Changes to either view, eg addition of a method or change of the class name, are reflected through to the other view. View (3) 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. The latter is controlled by event handlers managing the more complex consistency requirements.

Having defined a simple tool, and experimented with its notation, additional behaviour can be incrementally added using event handlers to implement more complex constraints. Examples include:

- type checking, e.g. UML associations must be between classes;
- model 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 modifying method entity properties in the modelling tool information model; or
- back end functionality, e.g. generating C# skeleton code from model instances.

These handlers can be generic for reuse (eg a generic horizontal alignment handler) or specific to the tool. As with other meta specification components, adding or modifying a handler results in “on the fly” compilation of handler code and incorporation of that code into any executing tool instances.

As noted above, back end functionality can be implemented by event handlers. In addition, as all tool and model components are represented in XML format, it is straightforward to implement back end processing

using XSLT or other XML-based transformation tools. This approach can allow back ends to be developed independently of the editing environment enhancing modularisation. An additional approach for implementing back end functionality is via Pounamu's web services-based API. This exposes Pounamu's model representation, modelling commands, menu extension capability, etc, permitting tight and dynamic integration of third party tools, and other Pounamu environments. We have, for example, used this API to implement peer to peer based synchronous and asynchronous collaboration support between multiple Pounamu environments, to implement generic GIF and SVG web-based thin client interfaces, to implement interfaces for mobile device deployment, and to integrate a Pounamu based process modelling tool with a process enactment engine [19].

Pounamu's extendibility also applies to the meta environment itself. For example, we have incorporated support for zoomable user interfaces as an alternative to the conventional editing interfaces described earlier. This extension uses the Jazz ZUI API framework [2] to implement a context and focus metaphor. Figure 10 shows an example of using these zoomable views in Pounamu. The Radar view (left hand side) provides a zoomed-out context for the diagram, allowing the user to zoom the radar view as a whole in and out. A zoomable view (middle, top) allows the user to selectively zoom individual shapes and a Split view similarly provides a selection from the Radar or Zoomable view that can be individually zoomed. Integration of the ZUI framework was simplified by using event handlers to manage and control changes in the ZUI views.

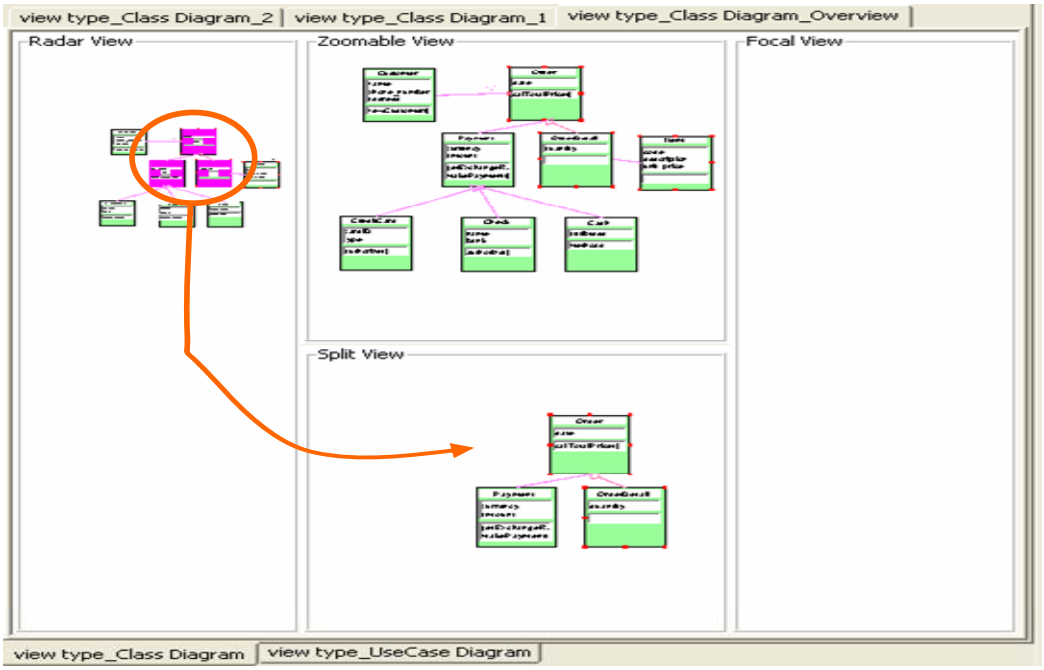

Figure 10. Zoomable views for complex Pounamu diagrams.

We have developed a wide range of exemplar DSVL tools with Pounamu, some of which are illustrated in Figure 11. These include:

- A full UML tool supporting all UML diagram types [48]. This also provides an import/export facility using the XML Model Interchange (XMI) standard allowing models to be imported from and exported from other XMI-compliant UML tools. A code generator takes XMI models from Pounamu and generates Java code that can be further extended by a programming environment.
- A circuit design tool (Figure 11 (a)) providing a CAD-like tool for circuit design.
- A web services composition tool, ViTABaL-WS [29] (Figure 11 (b)). ViTABaL-WS provides web service composition support to business analysts. It provides a tool abstraction composition view and a Business Process Modelling Language composition view both onto a shared model of business processes. The tool generates Business Process Execution Language for Web Services

(BPEL4WS) web service composition scripts which are run via a third-party workflow engine to realise the composed web service specification. The workflow engine uses Pounamu's web service API to support dynamic visualisation of enacted BPEL4WS models as a dynamic debugger.

- A statistical survey design tool SDLTool [24] (Figure 11 (c)). SDLTool provides multiple views describing statistical processes, data and analysis steps. This is used by statisticians to design, enact and process complex statistical surveys.
- A software process modelling and enactment tool IMAL [19] (Figure 11 (d)). IMAL provides multiple users multiple workflow modelling diagrams. These software process models can be enacted by the tool to help support work co-ordination by multiple developers. External tools are invoked via Pounamu's web service support to provide complex rule processing and XML document display and update. Pounamu itself is invoked by a workflow engine via Pounamu's web services API to support dynamic visualisation of enacted work processes.

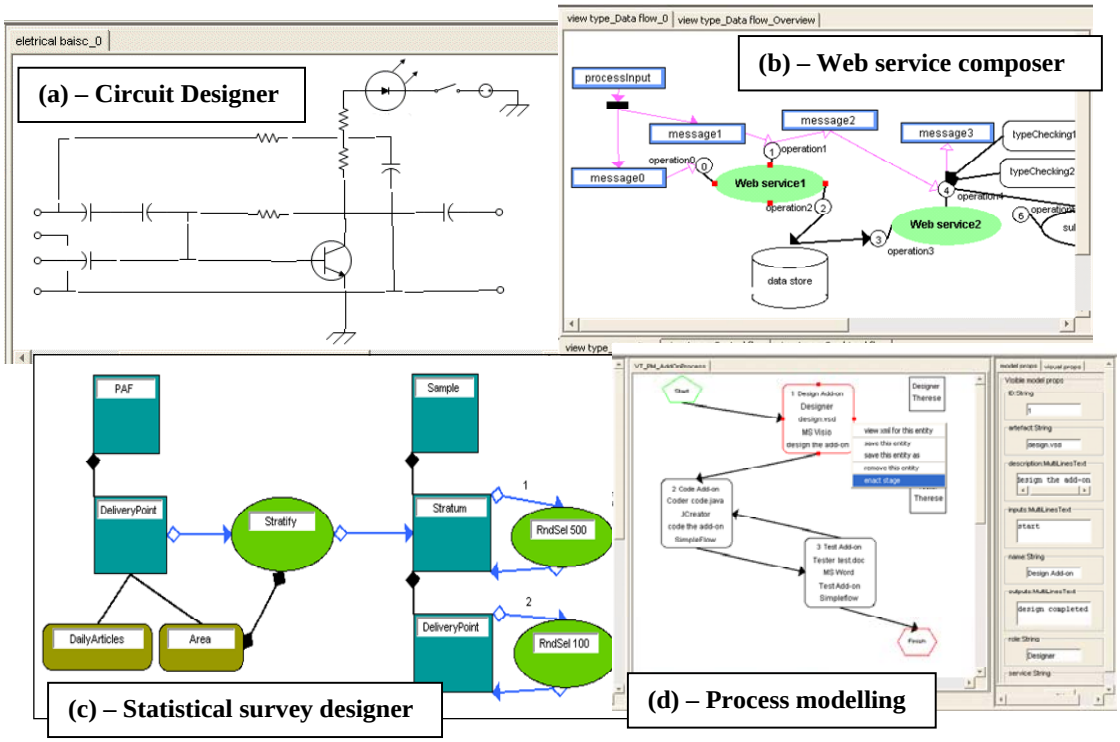

Figure 11. Examples of Pounamu DSVL tools.

We have also used Pounamu as a rapid prototyping tool in a range of industrial applications to assist in the design of visual notations and interfaces for client companies. These applications include: a business form designer; a business enterprise modelling tool; a project management tool with Gantt and work breakdown schedule views; and a web services composition tool. In each case the client companies were able to rapidly explore and evaluate a variety of alternative notational approaches with a low level of investment, hence allowing them to lower the risk of development.

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 use the web services API of Pounamu to extend *any* Pounamu-specified tool with:

- Editable web-based diagram views using either GIF or SVG (Scalable Vector Graphics) images
- Editable mobile PDA/phone diagram views using Nokia's MUPE framework
- Collaborative editing of diagrams

- Asynchronous version control and version merging support for diagrams, along with CVS repository management of versions

An example of the web-based, thin-client editing interface for Pounamu tools being used is shown in Figure 12 (left). This allows a group of users to interact with Pounamu views via web browsers and standard web software infrastructure. SVG image diagrams support browser-side drag and drop of diagram component using scripting.

A single Pounamu instance runs as an application server, while a set of Java servlets provide the web component implementation technology. No code changes are necessary to support this web-based diagramming, as the plugin is generic for any Pounamu specified tool; the servlets use the web services API of Pounamu to obtain available Pounamu tools and convert the view elements into GIF images or SVG encodings automatically.

We have developed an alternative set of software components using Nokia's MUPE framework that 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 MUPE XML mark-up for the view user interfaces. Figure 12 (right) shows an example of a project management Gantt chart diagram being browsed and manipulated on a mobile phone. The MUPE servers implement a flexible zooming mechanism which magnifies selected items dynamically to mitigate the small screen size of the mobile devices. Again, the plugin is generic, meaning that any Pounamu tool can be deployed using this technology without additional programming.

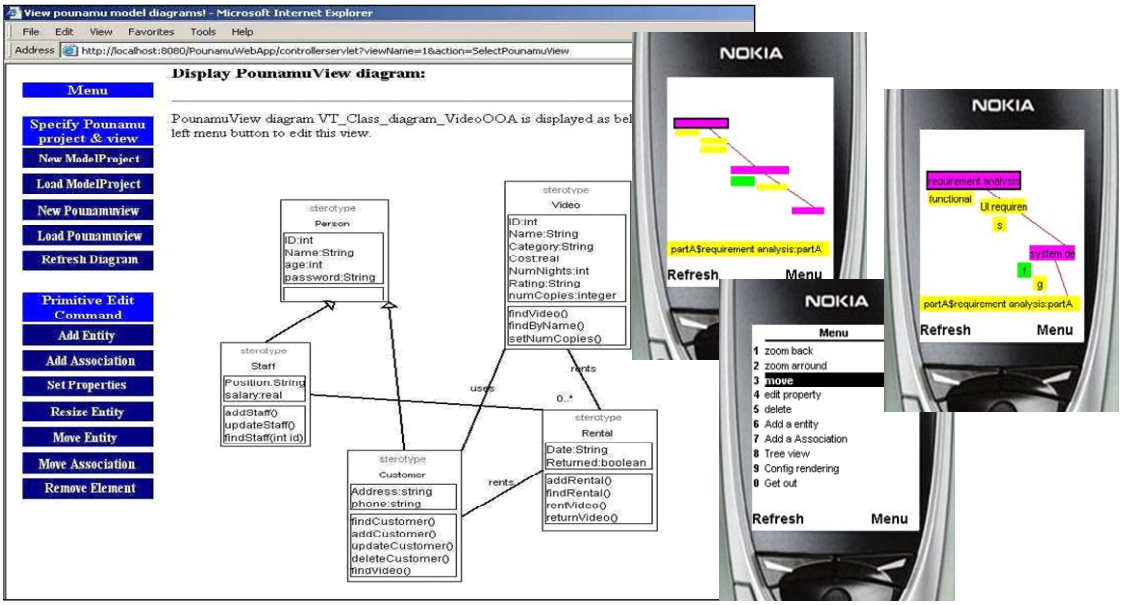

Figure 12. Thin client, web-based editing interface generated for Pounamu UML tool.

Collaborative work is supported in three ways: using one of the thin-client (web or mobile) user interfaces; using a set of plug-in synchronous editing components; or using a set of asynchronous version control and merging components. These all make use of Pounamu's web services API to support sending and receiving of collaborative editing messages between multiple Pounamu instances, synchronising views. They also support check in and check out of views from a CVS repository and visual differencing and merging support. An example of collaborative editing is shown in Figure 13 (a), where two users are editing a diagram together. Changes made by one user are sent to the other user's Pounamu and automatically applied. Changed diagram content is highlighted. A visual differencing algorithm uses a similar approach to support asynchronous version comparison and merging. An example of differencing two UML diagrams is shown in Figure 13 (b) where two versions have been compared and differences between them highlighted in the diagram.

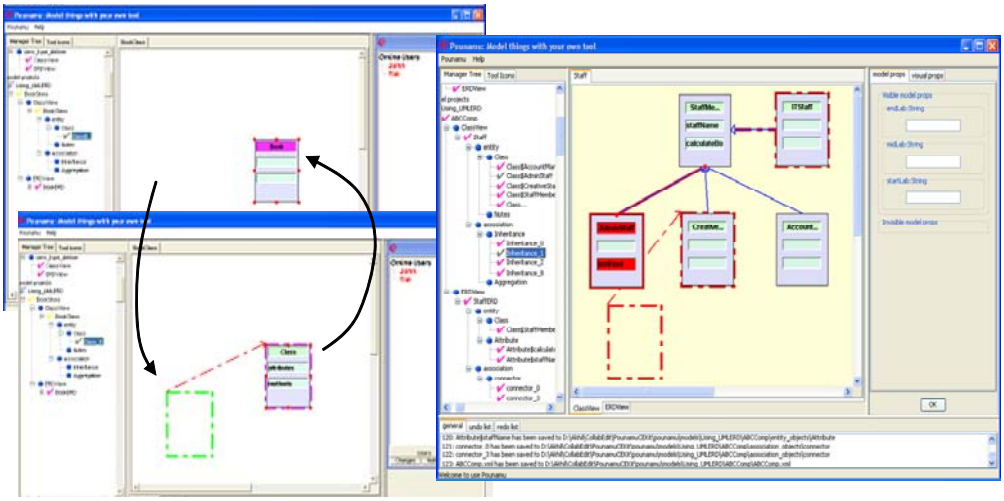

Figure 13. (a) Collaborative editing and group awareness; (b) version differencing.

Implementation

Pounamu is implemented in Java, using the Xerces XML parsing libraries, Swing user interface packages, and Java web services development toolkit. The main components of Pounamu are outlined in Figure 14. The Pounamu design and modelling tools use Java Swing to implement their user interfaces. The design tools create tool specifications, a set of shape, connector, view, entity, association, project and event handler types, which when composed together form a Pounamu tool specification. The modelling tool uses a model-view-controller architecture to provide data representation of entity, association, shape, connector and view instances (model); Swing-based representations of these model instances (view); and Command objects that modify the model state, created by interaction with the visual components (controller).

The Java JAX XML API and Xerces Document Object Model (DOM) framework were used for representing both tool specification data and modelling project data as in-memory XML data structures. The Java file management APIs were used for information storage and retrieval of tool specifications and modelling tool data to/from the XML format. A specialised class loader is used to support on-the-fly event handler class compilation and reloading, enabling dynamic code addition and replacement in the tool.

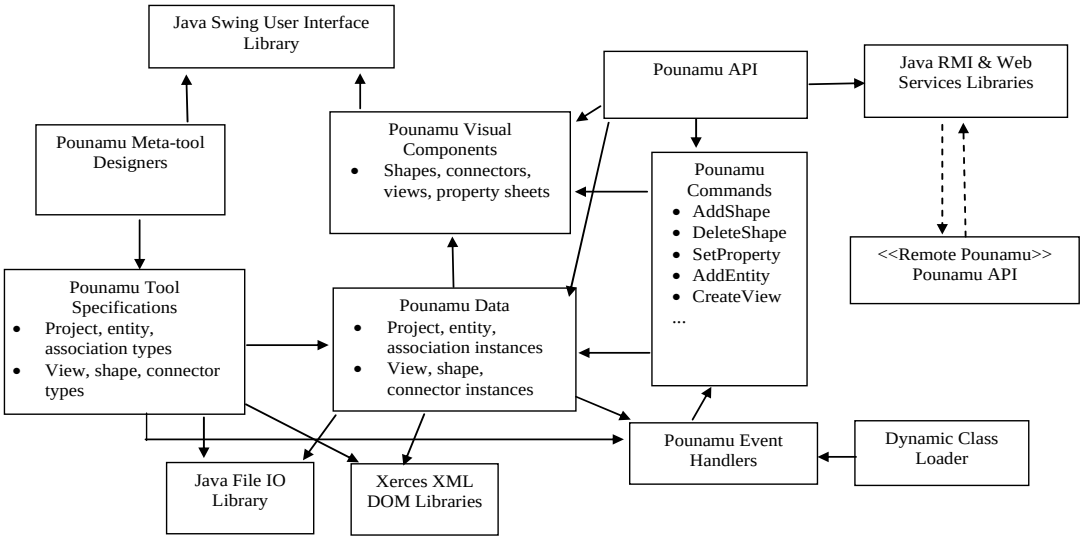

Figure 14. The component structure of Pounamu.

Specification and modelling tool information is represented in an XML format, both internally using a DOM API and externally in either XML files or a database. We chose an XML based representation to permit ready extension to the tool and model formats, ease of exchange with other tools, and the ability to use existing translation support tools. These XML formats also allowed us to adopt a web services-based API extension and integration approach for Pounamu. We have used XSLT translation scripts to support translation of Pounamu model and view data into other formats for information import and export [43].

The Java web services development toolkit was used to implement a web services API for Pounamu. This API provides external tools with the ability to query Pounamu model and diagram information in an XML format and to create and run Pounamu editing commands to update these data structures remotely. This web services API has been used to build the generic collaborative editing component for the Pounamu modelling tool which transmits XML-encoded diagram update messages between different instances of Pounamu. It has also been used to build the set of Java servlets used to provide the generic Pounamu thin-client interface plugin and a similar set of servlets for the MUPE-based mobile interface.

Evaluation

Evaluating a meta tool such as Pounamu is not a straightforward task due to the multiple points of view involved (tool developer, end user of developed tool, usability, utility, etc). Our approach has been to evaluate Pounamu at several levels and through a variety of mechanisms. These include:

1. Two large group experiments, spaced nearly two years apart, where participants (approximately 45 in each case) constructed a domain specific visual language tool and then surveyed. Our aim in each case was to use the feedback to improve the tool, and significant enhancement was undertaken between the two experiments.

2. Qualitative feedback, in the form of experience reports, from a smaller number of developers who used Pounamu to develop more substantial applications, such as the ones in Figure 11. These were used to assess whether perceptions altered as more substantial applications (with, for example, more complex back end integration requirements) were developed.

3. Small end user and cognitive dimensions [15] evaluations of the various Pounamu extensions, such as thin client and collaborative work support. These were primarily used to compare utility and usability of these extensions against the core Pounamu toolset. Many of these evaluations have been reported in detail elsewhere.

4. Small end user and cognitive dimensions evaluations of substantial applications developed using Pounamu. These were used to evaluate whether end users found Pounamu generated tools to have good usability characteristics. Many of these have also been reported in detail elsewhere.

Large group experiments

In each experiment around 45 participants, who were graduate-level students, were asked to construct a DSVL tool of their own choosing, but with at least a minimal set of required components, such as numbers of icons, views, handlers, etc so that tools with a realistic level of complexity were designed and constructed. Participants were given two weeks elapsed time (i.e. alongside other obligations) to complete application development; they were then surveyed using a set of open ended questions to qualitatively elicit strengths and weaknesses of Pounamu to construct the desired DSVL tool. The surveys, one undertaken in August 2004 (46 participants) and the second in May 2006 (45 participants), emphasised elicitation of weaknesses as their primary intention was to provide feedback to be used in improving Pounamu, hence the responses observed tended to describe generic strengths of Pounamu, with more detail on specific weaknesses.

Generic strengths emphasised by respondents in both surveys included: the rapidity with which tools were able to be constructed; the extensibility and customisability of the generated tools; the low learning curve needed to use Pounamu effectively; and the usefulness of being able to update tool definitions on the fly as iterative development was undertaken.

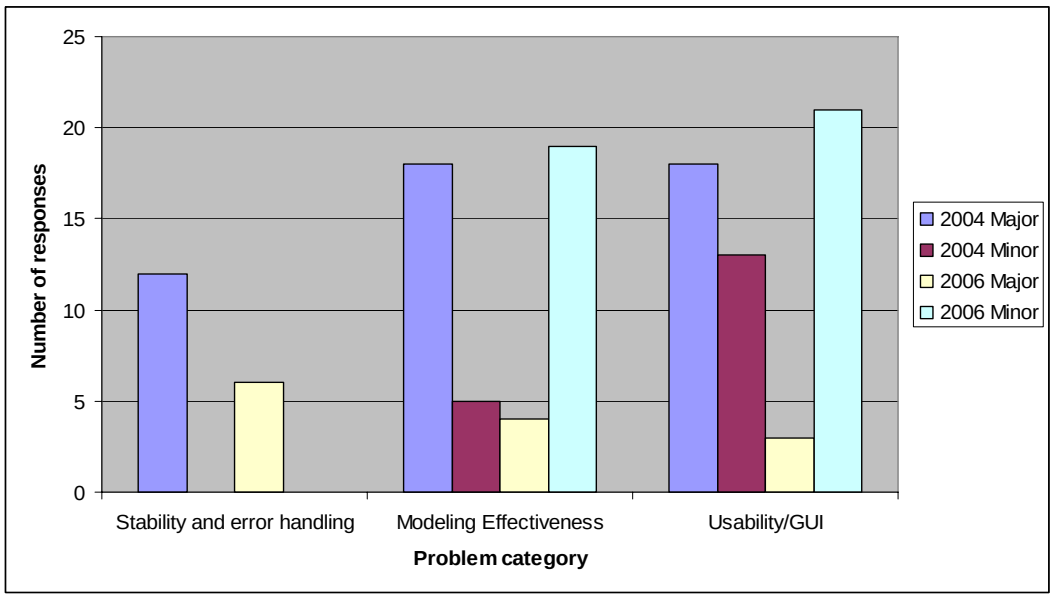

Figure 15: Problems identified in 2004 and 2006 surveys

Figure 15 charts the number of responses concerning identified weaknesses in each survey aggregated into three categories and subcategorised as "major", i.e. a significant weakness, or "minor", i.e. an issue causing irritation but not significantly affecting functionality. General weaknesses identified in the first experiment focused on issues of: stability of the software; inconsistency of the interface compared to other tools; lack of documentation, particularly for the API; difficulty of event handler specification; and weak error handling, all of which might be expected of the, at that stage, proof of concept prototype. Issues of stability, documentation, event handler specification and error handling were largely addressed in changes made to the system before the second experiment, which showed much less concern by participants on these issues. For each category the number of major problems identified was significantly lower in the second survey. Many minor usability issues were also identified by participants, such as the size of text fields for entering Java code for event handlers, some clumsiness around specification of icons, and the response time for some elements of functionality. The second experiment showed increased concern by participants in such issues, primarily, we speculate, because a lack of concern over major issues permitted them to focus more readily on more minor limitations of the meta-tool. Most issues identified concerned usability of the tool specification components, with very few issues concerns with the usability or efficacy of the generated tools themselves, nor of the representational power of the meta model.

A key feature for us is the efficacy of our evaluation-cycle based quality improvement approach which has demonstrated significantly enhanced user perception of Pounamu following attention to issues raised in the first large-scale user evaluation.

Large application developers

A smaller group of 8 developers have used Pounamu to develop more substantial applications, typically as an element of a larger research project, and over an extended period (several months at least). In each case, the developers provided detailed comments on the efficacy of Pounamu as part of a wider ranging implementation report. These qualitative comments were summarised and categorised in a similar manner to the large experiments. The small number of participants makes for less depth in the results, however, they are sufficient to assess any significant change in perception with increased application size. The general strengths identified were the same as for the large group experiments, but with more emphasis on the speed of development and the extendibility and customisability of the generated tools. Far fewer weaknesses were identified; familiarity with Pounamu appears to have mitigated many of minor difficulties identified in the large group experiments. Some issues around stability and performance were identified by those using Pounamu in its early stage of development, in common with the first group experiment, but those using later versions did not report the same issues. In summary, our results suggest that developers

using Pounamu to construct larger applications are somewhat more favourably inclined than those developing small applications as in the former case the benefits of efficiency of construction significantly outweigh minor usability issues.

Usability of Pounamu extensions

We conducted user surveys of the thin-client diagramming plug-ins (9 participants) and the collaborative editing plug-ins (10 participants) for Pounamu both using a UML modelling tool developed using Pounamu as a vehicle for the experiments. These are reported in detail in [6], [35]. Our primary aim was to assess efficacy of the extensions over that of the standard Pounamu thick client. Users performed similar design and revision tasks by themselves and in small (2-4 developers) groups. Feedback was generally very favourable, with users appreciating the ease of update of tools via the use of a shared web server and accessibility of diagrams via web browsers. The collaborative synchronous editing capabilities of Pounamu were found to be sufficient for basic revision and diagram construction tasks with the group awareness capabilities particularly praised by end users. The asynchronous versioning and merging support also received good feedback from users [35]. In addition to the usability evaluations, we performed a cognitive dimensions analysis of the versioning and merging plug-ins as well as assessing them against Gutwin's groupware framework [18].

Usability of substantial tools constructed using Pounamu

For many of the exemplar systems described earlier (and others) we have carried out a combination of survey-based end user evaluations of the application visual language environments and cognitive dimensions-based evaluations of the visual environment interaction and information presentation features. These evaluations have each been reported in detail elsewhere and include (in order of application development): the IMÅL distributed process modeling and enactment tool (reported in [19], 8 survey participants); a prototype UML tool (reported as a component of [35], 10 survey participants); and the SDL statistical survey specification tool (reported in [24], 8 survey participants). We summarize these evaluations here as evidence that tools implemented using Pounamu provide good usability characteristics.

We evaluated both IMÅL's modelling capabilities and its presentation of enacted process stage information in Pounamu modelling views [19]. Users reported they found the visual modelling facilities to be good, the multiple view support helpful, and consistency management between different views and error reporting to be good. However they found some modelling functions, such as resizing, to be ungainly, a lack of drag-and-drop creation of model elements, and some bugs in the modelling environment, all problems we have since addressed (IMÅL was developed using the same version of Pounamu as was used in the first group experiment). Our cognitive dimensions analysis of the process modelling tool found a good closeness of mapping of the visual notation to the problem domain, high consistency of views and few hard mental operations were needed to build models. However, at times there is insufficient visibility and juxtaposability. For example two views could not be seen at the same time. These findings have also led to a number of further enhancements to Pounamu, particularly in its view management facilities.

Our user survey of the prototype UML software design tool included experienced, industry modellers as well as novice users. We had users perform selected modelling tasks including creating new designs and revising designs. We also had users make small modifications to the UML design tool notation and meta-model using Pounamu's tool designers. Our Pounamu-built UML design tools were found to be both usable and appropriate to their task. Some problems were encountered with non-standard notational symbols and limited editing capability in some diagram types, especially UML sequence diagrams. These limitations arose from limitations in Pounamu's shape specification and editing event handler control. We have since enhanced both facilities and substantially improved the usability of the UML tools. Users found modifying the notational symbols used very straightforward with Pounamu's shape and connector designers. However they found modifying the meta-model and event handlers to be more difficult and desired improved support for these activities.

The SDL tool used the most recent version of Pounamu in its development and hence is most representative of the current tools state. Participants in the end user study were 8 senior undergraduate statistics students, who were given a tutorial introduction to SDL, prior to completing a series of tasks, followed by a survey including both closed and open-ended questions. Questions examined tool usability (90% positive) and

notation usability (75% positive), while observations of end user performance assessed diagram comprehension (75% good or better) and task completion (only 12% incomplete). Open ended responses focussed almost entirely on the statistical survey modelling ability of the tool rather than any usability or efficacy issues that could be related back to use of Pounamu. This was particularly pleasing as the discourse with participants ended up being entirely focussed on "real" end user requirements rather than technology difficulties imposed by the use of the metatool.

Assessing Pounamu against our two requirements, simplicity of use and simplicity of extension, we can conclude that Pounamu permits rapid development of a DSVL environment for a simple version of the supported notation, satisfying our first requirement. Many of the exemplar tools implemented with Pounamu have then been iteratively expanded in a manner matching the second of our requirements. For example:

- elaboration of notations, such as expansion of the range of UML diagrams supported in the UML tool has been carried out even while the tool has been in use, permitting exploratory DSVL development
- addition of event handlers for complex constraint management, particularly for visual constraints and for consistency management between elements in the information model. For example, our Traits modelling tool used this for generating a combined conflict free method list
- integration of backend code generation support has been provided for the UML, web services and process modelling tools. These have adopted approaches of transforming the saved Pounamu XML model structures into respectively Java code, BPEL4WS scripts, and XML data for display by Microsoft InfoPath.
- use of the web services API to integrate the process modelling tool with a distributed process enactment engine and Microsoft InfoPath.

The extended entity relationship based representation chosen for the tool information model, while simple, has so far proved adequate for most tools. In particular it was able to directly support implementation of the substantial UML meta model, itself a substantial subset of the OMG UML meta model [39], for the UML tool. The simple mapping representation supported by the view specification tool was adequate for most basic view-to-model data mappings with so far only a few, such as more complex UML and process flow diagrams, requiring more complex mappings implemented using complex event handlers. The visual event handler definer has reduced the need for end users to use the complex API-dependent Java coding of constraints for tools.

Summary

We have described Pounamu, a meta tool for specifying and implementing multiple-view multiple-notation diagramming tools. Our primary aim in developing Pounamu was to provide both ease of use and simple extendibility of both the meta tool and generated tools. These have been achieved through a simple conceptual base and a simple set of specification tools, together with live, incremental implementation, and a heavy emphasis on backend integration capability. The latter has permitted ready implementation of significant and generic extensions such as thin and mobile client deployment, synchronous and asynchronous collaboration support, and zoomable user interfaces, together with tool specific code generation and 3rd party tool integration implementations. We have used Pounamu to develop a broad range of DSVL applications both for academic/research use and industrial purposes and over a large end user base. Informal feedback and formal evaluation of the tools has been very positive.

In current work, we are developing a formula definer to augment Pounamu's meta-model and view designers, allowing tool developers to specify formulae over both model and view data structures combined with a one-way constraint system to compute values during tool usage. This will allow simpler specification of computed values and some forms of constraint handling within Pounamu tools. In addition we are developing a more complex view specification tool, allowing many-to-many mappings between view shapes and connectors and model entities and associations to be specified. This will again make it easier for tool developers to build more complex view-model mappings without resorting to using complex event-driven handlers. A set of plug-ins for the Eclipse open-source IDE to allow Pounamu tool specifications to be used to generate Eclipse graphical editors is under development, as are further thin-

client visualisation tools using VRML to produce 3D representations of complex Pounamu views, supporting complex design visualisation tasks.

References

[1] Baresi, L., Orso, A. and Pezze, M. Introducing Formal Methods in Industrial Practice. In *Proc. ICSE 1997*, Boston MA, 1997, ACM Press, pages 56–66.

[2] Bederson, B., Meyer, J. and L. Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java, in Proceedings of 2000 ACM Conference on User Interface and Software Technology, ACM Press, 171-180.

[3] Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. (1995): Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. Proc. of the 4th International WWW Conference, Boston, MA, December 1995.

[4] Buchner, J., Fehnl, T., and Kuntsmann, T., HotDoc a flexible framework for spatial composition, In Proceedings of the 1997 IEEE Symposium on Visual Languages, IEEE CS Press, pp. 92-99.

[5] Burnett M, Goldberg A, Lewis T (eds) *Visual Object-Oriented Programming*, Manning Publications, Greenwich, CT, USA, 1995.

[6] Cao, Shuping, Thin-client interface design for the Pounamu Meta-Case Tool, MSc Thesis, Department of Computer Science, University of Auckland, 2004, 164pp.

[7] Cypher, A. and Smith, D.C., KidSim: End User Programming of Simulations, In *Proceedings of CHI'95*, Denver, May 1995, ACM, pp. 27-34.

[8] 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.

[9] Ebert, J., Süttenbach, R., and Uhe, I., Meta-CASE in practice: a case for KOGGE, In *Proc. CAiSE'97, LNCS 1250*, 203-216.

[10] Eclipse, Graphical Editing Framework, www.eclipse.org/gef, accessed 2 October 2006

[11] Eclipse Graphical Modelling Framework, http://www.eclipse.org/gmf/, accessed 13 October 2006

[12] Ehrig, K., Ermel, C. Hänsgen, S. and Taentzer, G. Generation of Visual Editors as Eclipse Plug–Ins, *Proc. 2005 ACM/IEEE Automated Software Engineering*.

[13] 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.

[14] Gordon, D., Biddle, R., Noble, J. and Tempero, E. (2003): A technology for lightweight web-based visual applications, *Proc. of the 2003 IEEE Conference on Human-Centric Computing*, Auckland, New Zealand, 28-31 October 2003, IEEE CS Press.

[15] Green, T.R.G and Petre, M, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, JVLC 1996 (7), pp.131-174.

[16] Greenfield, J., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, http://msdn.microsoft.com/vstudio/DSLTools/ 2004.

[17] 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.

[18] Gutwin, C. and Greenberg, S., A Descriptive Framework of Workspace Awareness for Real-Time Groupware, Computer Supported Cooperative Work, vol. 11 no. 3, p.411-446, 2002.

[19] Helland T, A service oriented approach to software process support, MSc Thesis, Department of Computer Science, University of Auckland, 2004, 216pp.

[20] IBM Corp, Rational Rose XDE Modeler, http://www-306.ibm.com/software/awdtools/developer/modeler/

[21] Kaiser, G.E. Dossick, S.E., Jiang, W., Yang, J.J., Ye, S.X. (1998): WWW-Based Collaboration Environments with Distributed Tool Services, World Wide Web, vol. 1, no. 1, 1998, pp. 3-25.

[22] Khaled, R., McKay, D., Biddle, R. Noble, J. and Tempero, E. (2002): A lightweight web-based case tool for sequence diagrams, *Proc. of SIGCHI-NZ Symposium On Computer-Human Interaction*, Hamilton, New Zealand, 2002.

[23] 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.

[24] Kim, C.H., Hosking, J.G. and Grundy, J.C. A Suite of Visual Languages for Statistical Survey Specification, In Proceedings of the 2005 IEEE Conference on Visual Languages/Human-Centric Computing, Dallas, Texas, 20-24 September 2005, IEEE CS Press.

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

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

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

[28] Liu, N., Hosking, J.G. and Grundy, J.C. A Visual Language and Environment for Specifying Design Tool Event Handling, In *Proc. VL/HCC'2005*, Dallas, Sept 2005.

[29] Liu, N., Grundy, J.C. and Hosking, J.G. A Visual Language and Environment for Composing Web Services, In *Proc. 2005 IEEE/ACM Int. Conf. on Automated Software Engineering*, Long Beach CA, Nov 7-11 2005.

[30] Lyu, M. and Schoenwaelder, J. (1998): Web-CASRE: A Web-Based Tool for Software Reliability Measurement, Proc. of International Symposium on Software Reliability Engineering, Paderborn, Germany, Nov, 1998, IEEE CS Press.

[31] Maurer, F., Dellen, B., Bendeck, F, Goldmann, S., Holz, H., Kötting, B., Schaaf, M. (2000): Merging project planning and web-enabled dynamic workflow for software development, IEEE Internet Computing, May/June 2000.

[32] McIntyre, D.W., Design and implementation with Vampire, *Visual Object-Oriented Programming*. Manning Publications, Greenwich, CT, USA, 1995, Ch 7, 129-160.

[33] Mackay, D., Biddle, R. and Noble, J. (2003): A lightweight web based case tool for UML class diagrams, *Proc. of the 4th Australasian User Interface Conference*, Adelaide, South Australia, 2003, Conferences in Research and Practice in Information Technology, Vol 18, Australian Computer Society.

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

[35] Mehra, A., Grundy, J.C., Hosking J.G., A generic approach to supporting diagram differencing and merging for collaborative design, *2005 IEEE/ACM Automated Software Engineering*, Long Beach CA, Nov 2005.

[36] Microsoft Visual Studio 2005: Domain-Specific Language Tools, http://msdn.microsoft.com/vstudio/DSLTools/, accessed 13 October 2006.

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

[38] Myers, B.A., The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE TSE*, vol. 23, no. 6, 347-365, June 1997.

[39] Object Management Group, Unified Modeling Language UML Resource Page, http://www.uml.org/, accessed 2 October 2006.

[40] Phillips C, Adams S, Page D, Mehandjiska D, The Design of the Client User Interface for a Meta Object-Oriented CASE Tool, Proc TOOLS, 1998 Melbourne, p156-167.

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

[42] Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. 1996. Designing object-oriented synchronous groupware with COAST, *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM Press, November 1996, pp. 21-29

[43] Stoeckle, H., Grundy, J.C. and Hosking, J.G. Approaches to Supporting Software Visual Notation Exchange, *Proc HCC'03*, Auckland, New Zealand, Oct 2003, IEEE, 59-66

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

[45] Welch, B. and Jones, K. *Practical Programming in Tcl and Tk*, 4th Edition, Prentice-Hall, 2003.

[46] Younas, M.a.I., R. Developing Collaborative Editing Applications using Web Services. *Proc. 5th Int. Workshop on Collaborative Editing, Helsinki*, Finland, Sept 15, 2003.

[47] Zhang, K. Zhang, D-Q. and Cao, J. Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE TSE*, 27,4, April 2001, 289-307..

[48] Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, In Proc. VL/HCC 2004 Rome, Italy, 25-29 September 2004, IEEE CS Press, pp. 254-256.