A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools

Na Liu[1], John Hosking[1] and John Grundy[1, 2]

[1]Department of Computer Science and [2]Department of Electrical and Computer Engineering,
University of Auckland, New Zealand

{karen, john, john-g}@cs.auckland.ac.nz

Abstract

End users often need the ability to tailor diagramming-based design tools and to specify dynamic interactive behaviours of graphical user interfaces. However most want to avoid having to use textual scripting languages or programming language approaches directly. We describe a new visual language for user interface event handling specification targeted at end users. Our visual language provides end users with abstract ways to express both simple and complex event handling mechanisms via visual specifications. These specifications incorporate event filtering, tool state querying and action invocation. We describe our language, its incorporation into a meta-tool for building visual design environments, examples of its use and results of evaluations of its effectiveness.

Keywords: Visual Language, User interface, Event Handling, Meta Tool

1 Introduction

Visual design tools have many applications, including software design, engineering product design, E-learning and data visualisation. Pounamu (Zhu et al, 2004) is a meta-tool we have developed for building such visual design tools. High-level visual specifications of tool meta-models and visual language notations allow end users to modify aspects of their tools such as appearance of icons and composition of views. However, both our own and other researchers' experiences indicate that many end users also wish to modify tool behaviour (Morch, 1998; Peltonen, 2000) and reconfigure user interaction with their design tool. This includes specifying editing constraints e.g. diagram element layout; automated diagram modification e.g. auto-add or resize of elements; semantic constraints e.g. allowing connection of only certain typed elements; automatic computation e.g. calculating an attribute value from the values of connected diagram element attributes; and well-founded user interactions e.g. alerting users to invalid input.

Many end users of such tools are not programmers and do not wish to learn or use complex textual scripting languages to tailor their design tools in these ways. Most approaches for design tool tailoring, however, use just such techniques (Cypher and Smith, 1995; Lewicki and Fisher, 1996; Peltonen, 2000). Some tools support limited configuration via preferences and wizards, but these severely limit the tailoring possible (Morch, 1998). Programming by example has been used for end user configuration, but is limited in power and it is often hard to visualise and modify specifications learnt (Cypher, 1993; Smith et al, 1995).

Most visual design tools are "event driven", meaning when a user modifies a diagram in the tool, events are generated and can be acted upon to modify other diagram content, enforce constraints, etc. We have used the event-driven nature of such tools as a vehicle to provide end users with a domain specific visual language, *Kaitiaki*[1], with which to specify behaviours for their tools. We have added this visual language to our Pounamu meta-tool providing end users with little programming background, a mechanism to detect events and specify actions to take. We first motivate our work and survey related research, then outline our approach and its design and implementation. We finish with an evaluation, conclusions, and future work opportunities.

2 Motivation

Consider a diagram-based design tool for web site and GUI specification, an example of such is illustrated in Figure 1. This consists of a web site map view (rear) and a web form view (front). We have built this tool with our Pounamu meta-tool along with a many other diagram-based design tools (Zhu et al, 2004). Such applications allow end users to model complex design problems using visual notations appropriate to the domain. Pounamu allows us to specify the meta-model, shapes and views (diagrams) for tools such as these using a variety of visual languages. Pounamu tool end users can modify the specification, even while the tool is in use, and have their changes reflected in the running tool (Zhu et al, 2004). This is very useful for changing symbol appearance, adding new symbols and diagram types, and even extending a tool's meta-model. As many users of our tools are not programmers, providing ways of specifying behavioural changes is more challenging.

A variety of approaches have been used to support reconfiguration of diagramming tools. Frameworks, such as Suite (Dewan and Choudhary, 1991), Meta-Moose (Ferguson et al, 1999) and Unidraw (Vlissides and Linton, 1989) require modifications to the tool's code,

Copyright © 2007, Australian Computer Society, Inc. This paper appeared at the Eighth Australasian User Interface Conference (AUIC2007), Ballarat, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 60. Wayne Piekarski and Beryl Plimmer, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[1] *Kaitiaki* is the Maori word for handler, or guardian

with an edit-compile-run cycle. Some Tcl/Tk-based tools may be modified while in use (Welch and Jones, 2003), but this requires use of the Tcl programming language. MetaEdit+ (Kelly et al, 1996) and GME (Ledeczi et al, 2001) provide API based code integration facilities, but code must be pre-compiled. Usually only programmers familiar with the tool architecture can make such modifications.

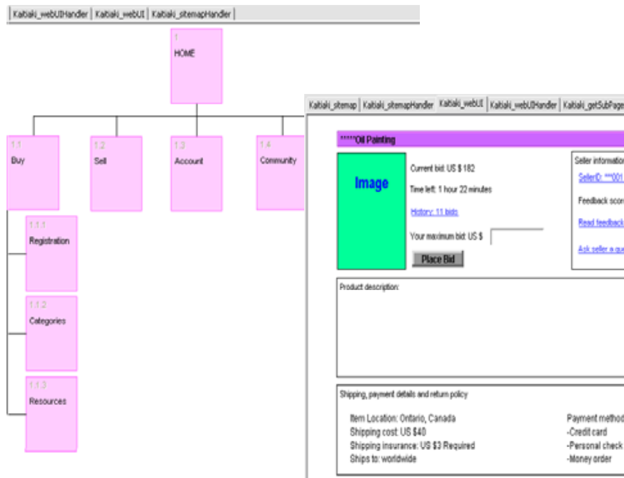

Figure 1. Example of a diagram-based design tool.

A common alternative approach supporting run-time modification is scripting. This is supported, for example, by Amulet (Myers, 1997) and Peltonen's UML tool (Peltonen, 2000). MetaEdit+ also provides a custom scripting language for report generation while GME uses OCL as a scripting language for constraint specification. These are difficult for non-programmer users to understand and use. Our Pounamu meta-tool uses this approach, with *event handlers* specified using textual Java fragments accessing a defined API and compiled on-the-fly. Figure 2 shows a Pounamu event handler for a web site design tool. This is a powerful mechanism for extending Pounamu and very sophisticated event handling behaviour has been implemented with it. While end users have been very complimentary of Pounamu's visual design tools, they have been less complimentary about the event handler specification as it requires programming skills and knowledge of the Pounamu API even for simple handlers.

Programming by demonstration and rule-based approaches have been used to specify behavioural constraints in some systems, often together and most notably in children's programming environments such as KidSim (Smith et al, 1995) and Agentsheets (Repenning and Sumnet, 1995). Most rule-based approaches exemplify "Event-Condition-Action" based visual languages where the user specifies an event of interest; conditions ("filters") when the action(s) should be run in response to the event; and action(s) to run to modify the tool's state.

Other Event-Condition-Action rule-based languages have been developed for a variety of domains, including building and tailoring design tools (Costagliola et al, 2002; Ledeczi et al, 2001; Lewicki and Fisher, 1996), user interface event handling (Berndtsson et al, 1999; Jacob, 1996), process modelling (Grundy et al, 1998) and database rule handling (Matskin and Montesi, 1998). However these approaches often suffer from use of inappropriate, textual rule-based languages for end users; reliance on many abstract concepts like control structures and variables; limitations on expressive power of the languages; difficulty in visualising and debugging learned rules from demonstration by the user; and limitations of reconfiguration power, including compile-time rather than run-time changes.

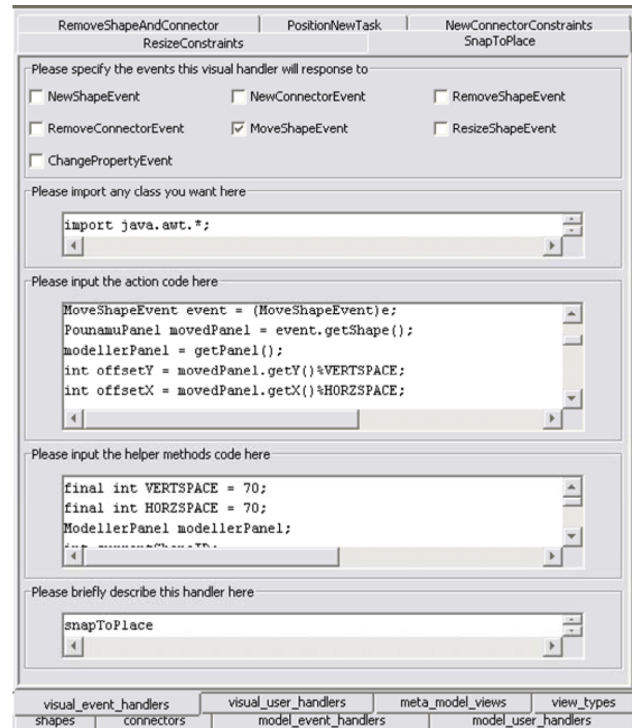

Figure 2. Example of event handler textual specification

3 Our Approach

Given the problems noted above, we wanted to replace Pounamu's textual, Java code-based event handler specification tool with one using a visual language suitable for non-programmer end users. To develop this replacement visual language, *Kaitiaki*, and its specification tool we carried out an analysis of Pounamu event handlers from a wide range of tools to identify key constructs used to specify different tool behaviours. All had aspects of (1) specifying the event(s) of interest; (2) querying the tool state in various ways; (3) filtering event/query results and making decisions; and (4) performing state changing actions on filtered objects. We also looked at the metaphors used in existing rule-based and event-condition-action event handler specification tools to see how these manifested the behavioural specifications and how suitable these were for end users. From this analysis and survey, we developed a set of key requirements and design approaches for our new *Kaitiaki* visual event handler designer:

- A need to represent key "building blocks" of state query, data filtering and state modification (actions).

- A need to represent event objects and their attributes; various objects from the Pounamu tool state (both view and model); and query results (typically collections of Pounamu state objects).
- A need to represent "data" propagation between event, query, filter and action representations.
- A need to represent iteration and conditional data flow.

The metaphor used by *Kaitiaki* is thus an "Event-Query-Filter–Action" (EQFA) model. This is articulated as "When this event happens, I want these changes made to these things". This is loosely based on our Serendipity event handling language which has been successfully used by end users in the process enactment domain to express similar kinds of event-driven behavioural models (Grundy et al, 1998). The key visual constructs of our language are representations of events, tool objects, queries on a tool's object state, state changing actions (including primitives relevant to common event handler requirements), and data flow links between these.

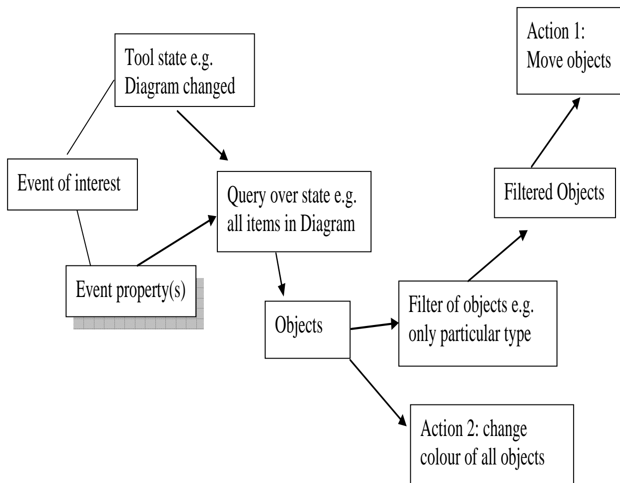

Figure 3. The Kaitiaki EQFA metaphor.

A *Kaitiaki* event specification is conceptually of the form outlined in Figure 3. An end user selects an event type of interest; adds queries on the event and Pounamu tool state (usually diagram content or model objects that triggered the event); specifies conditional or iterative filtering of the event/tool state data; and then appropriate state-changing actions to be performed on target tool state objects.

Complex event handlers can be built up in parts and queries, filters and actions can be parameterised, and reused. Ordering is handled by dependency analysis in the code generator. Domain specific tool icons are also incorporated into the visual specification of event handling as placeholders for the Pounamu state, to annotate and make the language more expressive.

4 *Kaitiaki* Visual Notation

The design of our *Kaitiaki* visual language focuses on supporting modularity and explicitly representing data propagation. We have avoided using abstract control structures and adhered to a dataflow paradigm to reduce the user's cognitive load. An overview of the main constructs of *Kaitiaki* is shown in Table 1 with an example *Kaitiaki* event handler view shown in Figure 5. From this we see the visual form of the constructs described in the previous section, i.e. events, filters, tool state queries, and actions plus iteration over collections of objects, dataflow input and output ports and connectors, and concrete iconic forms.

A single event or a set of events is the starting point for a *Kaitiaki* event handler specification. From this event various data flows out (event type, affected object(s), property values changed etc). Queries, filters and actions are parameterized with data propagated through incoming connectors. Multiple flows are supported with multiple dataflow connectors pointing to/from a visual construct. Queries retrieve elements and output one or more data elements; filters select elements from their input; actions apply operations to elements passed to them.

Event representation	EventType
Abstract Pounamu state representation	Single Data Element Collection of Data Elements
Filter	FilterName
Query on a tool's state	query
State changing action	ActionName
Iteration	for-each condition
Data propagation link	
Data flow ports in and out	
Concrete specification of Pounamu model elements (state)	HOME etc.

Table 1. Kaitiaki language key visual constructs.

State querying	
shape name getProp propValue	Obtain a named property value of a shape
getShapes shapes	Obtain all the shapes in the modeller panel

getConnectors connectors	Obtain all connectors in the modeller panel
shape getShapeConnectors connectors	Obtain all connectors connected to a shape
Data filtering	
shapeType	Select shapes of type from set or test type of single data element input
connectorType	Select a given connector type
shape shapes connectedTo shapes	Select all shapes that are connected from a particular shape (i.e. connector source)
shape shapes connectedFrom shapes	Select all shapes that are connected to a particular shape (i.e. connector target)
!null	Filter on a not null value
true	Filter on an expression value
State modification	
shape name value setProps	Set a list of name-value pair properties for a shape
name value setProps	Set a value to a named property
name values setListProp	Set a list of values to a named property
shape moveBy deltaX,deltaY shape moveTo X,Y	Move a shape by an offset to a location
shapes alignH alignV shapes	Horizontally/vertically align a shape with other aligned shapes
type newShape	Create a new shape
shape1 shape2 connectorType connect	Create a connector of a specified type and connect two shapes using the connector

Table 2. Overview of Kaitiaki reusable building blocks.

Queries and actions are invoked immediately when their actual data parameters are available (data push). If no related data dependency is specified, i.e. no data input parameter flows to the constructs, then queries and actions are invoked on demand when all other parameters to a subsequent flow element have a value (data pull).

Table 2 shows some of the predefined primitives for these constructs. These define the core vocabulary for our domain specific language, providing a base set of operations useful for diagram and diagram element manipulation. Typically this involves locating or creating elements, setting their properties, relocating/aligning them, and connecting them.

5 Examples of *Kaitiaki* Specifications

To construct a visual event handler specification a user identifies the target affected shape, view or model entity. She specifies the event(s) the event handler should respond to, and then adds building blocks to the handler specification. The concrete representations of Pounamu data, such as the shape icons, allow her to relate her queries, filters and actions to concrete objects in Pounamu. Basic elision support lets her show and hide concrete icons, queries, filters and actions to help manage larger specifications. To better illustrate the expressiveness of *Kaitiaki*, we use as examples several event handlers defined for the web site design tool shown in Figure 1. The web site map view (of a simple model like eBay) supports a hierarchical breakdown of web pages for sub-paging management. It requires several layout constraints to be enforced.

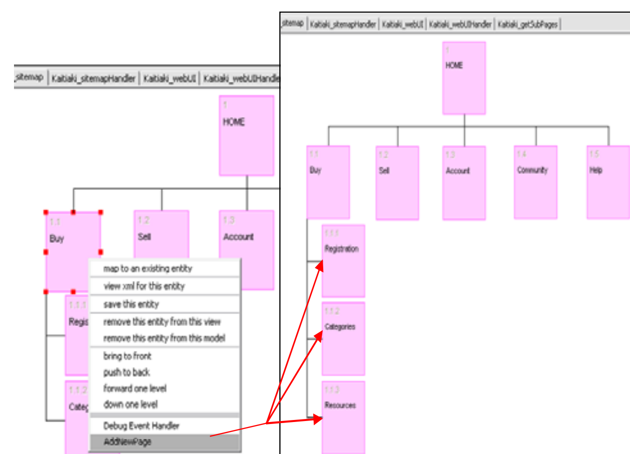

Figure 4. Example of addition of a new sub-page.

5.1 A Layout Constraint Event Handler

When creating a page icon for the web site map diagram, several values for its properties need to be set. These are gathered from a range of sources. An event handler is needed to implement one of the layout constraints. Users need to be able to create a new page by a right-click on an existing page; the newly created page is made a child of the existing page and a link is drawn between the old and new pages. The new sub-page and all other sub-pages belonging to this parent are aligned and repositioned upon arrival of the new page. Figure 4 shows the effect of this

event handler when a new sub-page is added to the selected.

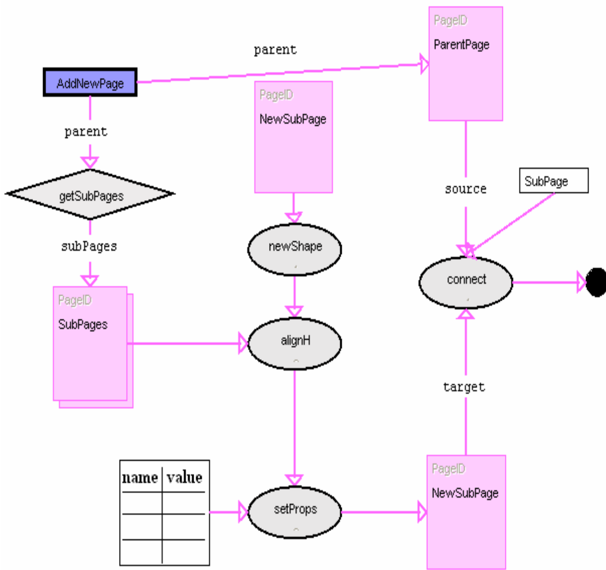

Figure 5. Specifying a layout constraint event handler.

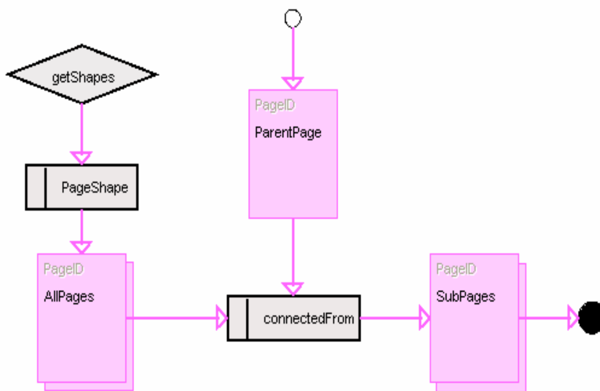

Figure 6. An example of a reusable visual query.

The event handler specification for this task is shown in Figure 5 which demonstrates the use of predefined Kaitiaki primitives (e.g. create, align and set property and connect shapes). It also demonstrates package and reuse of queries and actions. The modelling constructs contained in this event handler specification include a user defined trigger event (a context-menu event) called *AddNewPage* which has the acting *Shape* flowing from it; a query, *getSubPages* (a packaged query) that locates existing sub-pages of the currently selected page shape (*parent* as propagated to the query); four actions, the *newShape* action creates the new page shape; the *alignH* action does a horizontal alignment (with a user specified vertical distance in between) of the new page shape with the other existing sub-pages; the setProps primitive then sets default properties for the newly created page shape; and the *connect* primitive creates a connector of the *SubPage* connector type and connects the new page shape with its parent shape using the connector, now the event handler leads to a final stage, i.e. the end of the event handler specification.

Data sourced from outputs of "source" entities flows through data propagation links to act as input to "sink" entities. Each of the data propagations is statically checked for type compatibility of their data sender and consumer. Also incorporated in the event handler example are some end-user target tool icons, e.g. one on the flow from the *AddNewPage* event to the connect action annotates the flow to visually indicate the type of shape (page) on the flow. Another on the flow from the *setProps* action annotates the flow to indicate that the state change (which sets defaults values) also affects a page shape (the new sub-page). Shadowed icons, such as the one on the *subPages* flow from *getSubPages*, indicate multiplicity in the result. These optional annotations do not affect the semantics and are secondary notation augmenting the specification (although their types are checked). They have been generic titles (*ParentPage, NewSubPage,* etc) to emphasise the reusability of the event handler for other page shapes.

Figure 6 shows the packaged *getSubPages* query, which is composed of a number of primitives. We explicitly specify start (data flow in) and end (data flow out) ports for a package. Starting with a parent shape flowing in from the start to the *connectedFrom* filter, the *getShapes* query which gathers all available shapes (via data pull) is invoked. The *PageShape* filter selects all shapes that are of the *PageShape* type. The *connectedFrom* filter then selects only those that are connected from the specified parent shape. The end flow of the composed query indicates that on termination, this query flows out the set of sub-pages of the parent page. This query is invoked in the event handler in Figure 5, but can be reused by other event handlers. Actions and filters can similarly be specified and reused.

5.2 A User Interaction Event Handler

Kaitiaki also supports specification of user interface event handlers which can be used to add event-driven backend extensions such as generating JavaScript for a dynamic web site. Kaitiaki provides a set of frequently used form-based visual primitives to specify such interactions, including user interface rendering, e.g. adding a control or setting focus; form content modification, e.g. inserting web content or clearing selections; content validation, e.g. field format checks; confirmation prompts, e.g. to permit proceeding to a next step; error notification; and page navigation.

An example of the web form design view is shown in Figure 7. This includes a web form design (1) in the style of an eBay auction site including labels, buttons, hyperlinks, images etc. An event handler specifies behaviour triggered by a button *OnClick* event (2); when the *PlaceBid* button is clicked, the input the user has entered in the textbox is to be validated. If input is invalid, forward navigation is prohibited, the textbox cleared and focus set to the textbox so the user can re-enter a valid input; a validation message is also set to guide the user through the process. JavaScript is generated from the visual event handler specification and is inserted into the client-side script source.

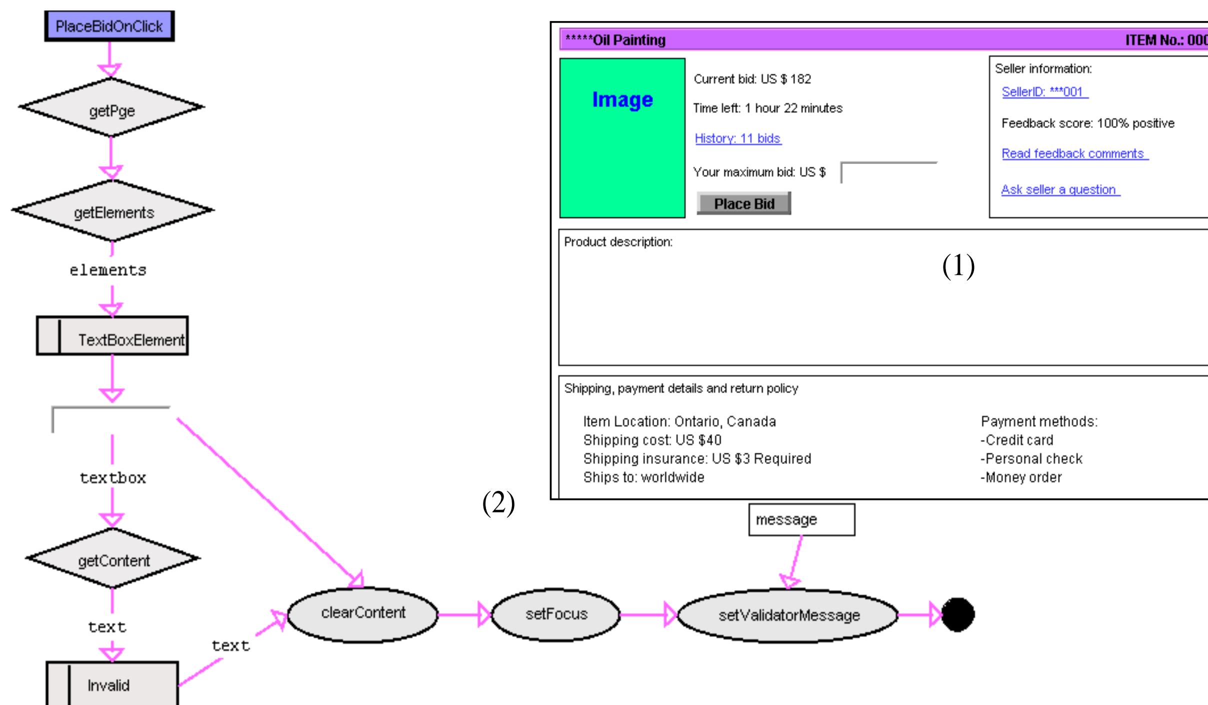

Figure 7. Specifying a user interaction event handler.

5.3 Dynamic Visualisation of Event Handlers

A consequence of introducing a visual language to generate Pounamu event handler code from visual specifications is the need to support their incremental development and debugging. To this end we have developed a visual debugger which dynamically annotates an event handler specification view for a fired event. The viewer exploits the dataflow between event handler building blocks to update a visualization of event handler execution in its own view.

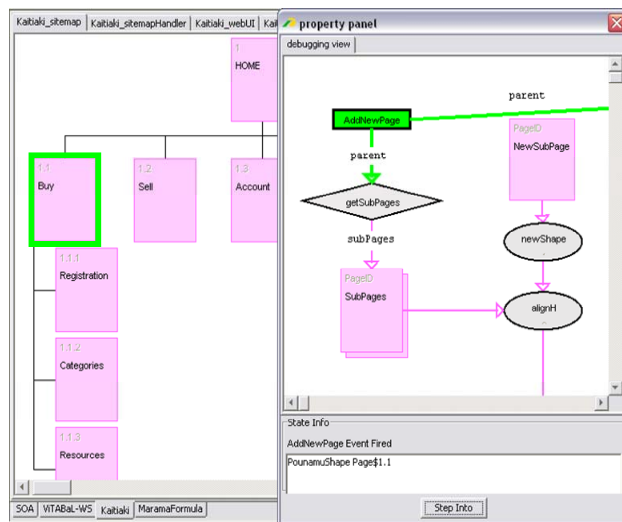

Figure 8. Visualising execution of a visual event handler.

The dynamic visualization of an event handler execution includes the visualization of EQFA element invocation (by flashing the corresponding node in the graph) and the visualization of data propagating path to the next construct (by highlighting the dataflow path). The traditional "debug and step into" metaphor is used and step-by-step visualization controlled by menu command. As seen in Figure 8, when the "Step Into" button is clicked, the next element to be invoked and the data propagation path are highlighted and handler execution pauses. The user can then step into the next element, abort the handler or inspect data values on a propagation path. The final state of the event handler execution highlights all the invoked constructs (nodes coloured in green) and the entire data propagation path. The states of the propagated data are able to be displayed in the debugging state information panel.

6 Design and Implementation

We have implemented an environment for *Kaitiaki* as an extension to the existing Pounamu meta tool. As shown in Figure 9, the main components added to Pounamu to generate Pounamu event handling code and visualize a running event handler include: Pounamu views and model for specifying visual event handler models; XML-based representation and storage for both library and user-defined queries and actions; and the visual debug viewer.

We have developed form-based specifiers for queries and actions to allow reconfiguration/modification of existing library code modules and creation of new ones by expert users. These are added to the library of reusable building blocks so end users can visually add them to specifications. Query/Action XML DTDs have been defined for Pounamu and XML data files are used for saving to and loading from a library of queries and actions. Visual *Kaitiaki* nodes are integrated with code modules by the code generator. There is strong coupled mapping of visual components and code components,

thus component-based code generation from a specification is achieved. The visual links (connectors) instantiate the visual entity components as they are required i.e. initialise query/action modules and invoke them as needed. The independent use of component-based visual and code components increases the modularity and reusability of the programming constructs.

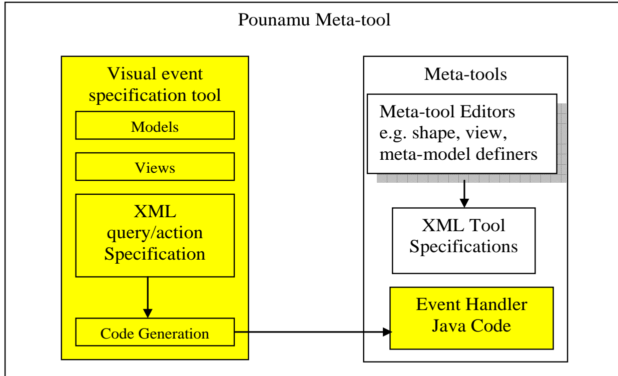

Figure 9. Extensions to Pounamu (highlighted).

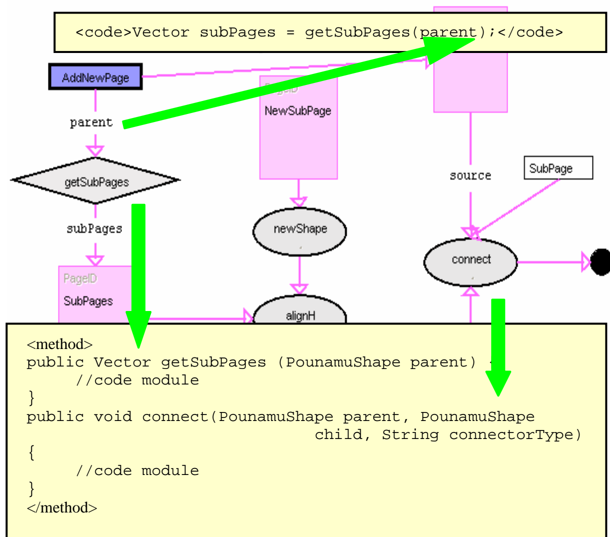

Figure 10. Compiling a visual event handler.

The code generator first performs a model (dependency) analysis and then sets module properties obtained from the visual model. It buffers code for creating event instance and query/action invocation, and finally writes the completed event handler code to an XML file. Figure 10 shows an example of this translation for the *AddNewPage* specification. Data propagation links instantiate actual method calls to target queries or actions, generating the `<code>` XML construct in the Pounamu event handler XML. Each query and action pulls out the reusable, parameterised code from the component. Parameter values are substituted and XML is generated for the `<method>` construct of Pounamu in the event handler XML.

7 Discussion and Evaluation

We have carried out a Cognitive Dimensions (Green et al, 2000) investigation of our visual event specification language and prototype environment to gauge its effectiveness. This identified the following features:

Abstraction gradient - Abstractions introduced are visual iconic constructs and data flow between them. These abstractions support query/action composition allowing users to specify Pounamu data queries and state changing actions as discrete, linked building blocks in the language.

Closeness of mapping - *Kaitiaki* constructs map onto the basic features of our EQFA metaphor. Concrete representation of Pounamu data elements is supported too. The metaphor is related to the way Pounamu supports event processing and mixes abstract and concrete constructs.

Error proneness - The existing Java-based Pounamu event handler designer is very error-prone for both novice and experienced users due to reliance on API knowledge and Java coding. *Kaitiaki* reduces some areas of error proneness by hiding API details and using data flow and visual constructs. However, as the specification is still an abstraction users can still specify faulty behaviour.

Progressive evaluation - *Kaitiaki* allows progressive evaluation of a visual event handler specification even when it is partially complete. Modifications to event handlers take effect immediately after re-registration in an end user tool. The visual debugger allows a user to step through a handler's elements and view data, which isn't supported by the Java code based event handler.

Viscosity - Modifying an event handler specification is by direct manipulation and a user can change one module's without affecting the rest of the specification.

Hard Mental Operations - The dataflow metaphor and visual constructs used as primitives in *Kaitiaki* increases its comprehensibility compared to the Java-based version.

Secondary Notation - *Kaitiaki* allows the user to layout, resize and annotate items in the view with iconic and textual labels, to increase a specification's readability.

Visibility and Juxtaposability - Information for each element of an event handler is readily accessible. The visualization of a running event handler is juxtaposed with the modelling view that triggers its execution.

Consistency and Hidden Dependency - Hidden dependency is introduced in both Pounamu and its specified tools to manage consistency in multiple views.

An informal evaluation of the visual event handler specification tool has been carried out with experienced Pounamu users and some novice users. Feedback suggests the visual specification approach is greatly favoured for most event handler specification tasks. We plan a more formal evaluation with novice users to better gauge this.

With respect to requirements, our EQFA metaphor captures event generation, state querying, filtering and iteration over query results, and state change actions to describe event handler specifications. The dataflow metaphor describes the composition of these event specification building blocks and seems to map well onto users' cognitive perception of the metaphor. Packing complex parts of a specification into reusable building blocks allows very complex event handlers to be defined with the model. A proof of concept support tool has

demonstrated the approach is feasible permitting both simple and complex Pounamu event handlers to be defined visually, code to be generated for them and visual debugging of them supported.

A potential weakness of *Kaitiaki* is the abstract representation of all events, queries, filters and actions. We have attempted to mitigate this with the addition of concrete iconic representations and are experimenting with elision techniques that allow concrete icons and *Kaitiaki* elements to be collapsed into a single meaningful icon.

8 Summary

We have developed a prototype visual language and proof-of-concept support environment for specifying diagramming tool event handlers. This uses a metaphor of generating event, tool state queries, filters over query results and state changing actions, with dataflow between these building blocks. The support environment allows users to compose handlers from these constructs and relate them to concrete diagramming tool objects. A debugger uses the visual notation to step through a specification, animating constructs and affected diagram objects. We have added this tool to the Pounamu meta-diagramming tool and specified and generated event handlers for example tools, demonstrating the feasibility of the approach.

We are exploring a programming by example extension to allow users to make changes to an existing modelling tool view and generate actions and data flow connections between actions in an event specification view. These are then tailored and abstracted by adding queries and filters to make a generic event handler. The dataflow metaphor used to compose a specification has interesting potential concurrency issues for parallel flows. We are examining extra synchronisation constructs to manage this. In addition, automatic layout of an event handler specification may be useful to improve a user's ability to show/hide/ collapse parts of a specification to manage size and complexity.

9 References

Berndtsson, B., Mellin, J., and Hogberg, U. (1999): Visualization of the Composite Event Detection Process. Proc Intnl Workshop on User Interfaces to Data Intensive Systems, IEEE CS Press, pp. 118-127.

Costagliola, G., Deufemia, V., Ferrucci, F., Gravino, C. (2002): The Use of the GXL Approach for Supporting Visual Language Specification and Interchanging. Proc HCC'02, pp131-138.

Cypher, A. and Smith, D.C. (1995): KidSim: end user programming of simulations, Proc CHI'95, pp. 27-34.

Cypher, A. (1993): Watch What I Do: Programming by Demonstration, MIT press.

Dewan, P. & Choudhary, R. (1991): Flexible user interface coupling in collaborative systems, CHI'91, 41-49.

Ferguson R, Parrington N, Dunne P, Archibald J, Thompson J ((1999): MetaMOOSE-an object-oriented framework for the construction of CASE tools: Proc CoSET'99, LA.

Green, T. R. G., Burnett, M. M., A Ko, J. (2000): Rothermel, K. J., Cook, C. R., and Schonfeld, J., Using the Cognitive Walkthrough to Improve the Design of a Visual Programming Experiment, Proc VL2000, 172-179

Grundy, John, Rick Mugridge and John Hosking (1998) Visual Specification of Multi-View Visual Environments. Proc VL'98, 236-243.

Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. (1998): An architecture for decentralised process modelling and enactment, IEEE Internet Computing, 2:5, 53-62.

Jacob, R. A (1996): Visual Language for Non-WIMP User Interfaces, Proc. VL'96.

Kelly, S., Lyytinen, K., and Rossi, M. (1996): Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, Proc CAiSE'96, LNCS 1080.

Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G. (2001): Composing Domain-Specific Design Environments, Computer, 44-51.

Lewicki, D. and Fisher, G. (1996): VisiTile - A Visual Language Development Toolkit, Proceedings of the 1996 IEEE Symposium on Visual Languages, pp. 114-121.

Matskin, M. and Montesi, D. (1998): Visual Rule Language for Active Database Modelling, Information Modelling and Knowledge Bases IX. IOS Press, pp. 160-175.

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

Morch, A. (1998): Tailoring tools for system development, Journal of End User Computing, pp. 22-29.

Myers, B.A (1997): The Amulet Environment: New Models for Effective User Interface Software Development, IEEE TSE, vol. 23, no. 6, 347-365.

Peltonen, J. (2000): Visual Scripting Language for UML-based CASE tools, Proc. of ICSSEA 2000, volume 3.

Repenning, A. and Sumnet, T. (1995): Agentsheets: a medium for creating domain-oriented visual languages, Computer, 28, no. 3.

Smith, D.C., Cypher, A. and Spohrer, J. (1995): KidSim: programming agents without a programming language, Communications of the ACM, vol. 37, no. 7, pp. 54 – 67.

Vlissides, J.M. and Linton, M. (1989): Unidraw: A framework for building domain-specific graphical editors, Proc. UIST'89, ACM Press, pp. 158-167.

Welch, B. and Jones, K. (2003): Practical Programming in Tcl and Tk, Prentice-Hall.

Zhu, N., Grundy, J.C. and Hosking, J.G. (2004): Pounamu: a meta-tool for multi-view visual language environment construction, Proc VL/HCC'04, pp. 254-256.