In Proceedings of 2nd Component Users' Conference, Munich, July 14-18 1997.

A Visual, Java-based Componentware Environment for Constructing Multi-view Editing Systems

John Grundy

Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

Rick Mugridge and John Hosking

Department of Computer Science
University of Auckland
Private Bag, Auckland, New Zealand
{john, rick}@cs.auckland.ac.nz

Abstract

We describe a collection of tools supporting the development of multiple view, JavaBeans based environments. The appearance and interactive behavious of graphical user interface components and editors is specified visually using BuildByWire, which generates JavaBeans component implementations. Multiple view, view consistency and cooperative work support is provided using the JViews toolkit, which provides specialisable classes for component repository and inter component consistency relationships. JComposer allows JViews components and relationships to be specified visually and combined with BuildByWire visual components, editors and third-part Java Beans to generate substantially complete multiple view environments.

1. Introduction

In recent work we have developed a wide range of multi-view editing systems, including User Interface builders [Mugridge et al 1996, Hosking et al 1995, Grundy et al 1996a], CASE tools [Grundy and Hosking 1996], Software Engineering Environments [Grundy et al 1995], CAD/CAM systems [Amor et al 1995], Groupware tools [Grundy et al 1996b], and process modelling environments [Grundy et al 1996b, Grundy and Hosking 1996]. For example, Figure 1 shows an example from SPE-Serendipity [Grundy et al, 1996b], which is an integrated software development and process modelling environment.

To date, we have used an object-oriented toolkit, MViews, [Grundy and Hosking 1996] to implement such systems, and have not used component-based solutions. However, it was apparent that our systems could make good use of componentware to improve reuse and integration of individual editor and repository components, consistency management techniques, and indeed the whole editors and tools themselves. Existing componentware toolkits, such as OpenDoc [OpenDoc 1996], ActiveX [Active X 1996] and JavaBeans [JavaBeans 1996], however, lack sufficiently flexible inter-component consistency mechanisms and component linkage abstractions for our purposes [Grundy et al 1996a]. They also lack suitable development tools for the rapid construction of such environments. While tools are available for simple form based component construction [Borland 1997, Powersoft 1997], there is little support for the direct manipulation of visual notations, as used in our environments.

To address this deficiency, we are developing a Java-based componentware toolkit, with direct-manipulation interface and repository component builders, which supports the development of multi-view applications. This consists of three components:

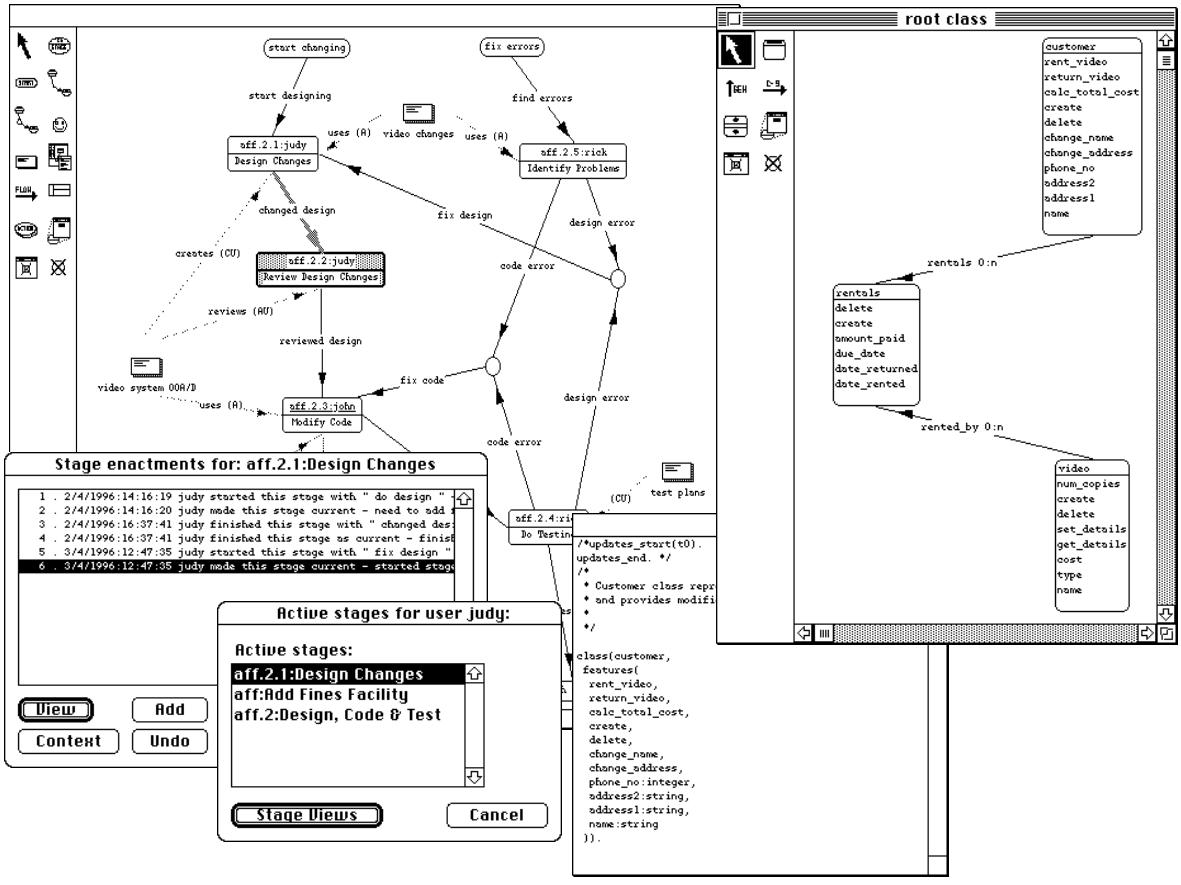

Figure 1. An example of our previous work in developing multi-view, multi-user environments.

- *BuildByWire*, a direct manipulation editor for specifying the user interface components and basic layout behaviour for visual notations, together with editors for constructing and manipulating diagrams using the notations.

- *JViews,* a componentware toolkit for designing and implementing the data, behaviour and inter-component consistency management aspects of view and repository components. JViews view components are interfaced to BuildByWire components to form a multi-view, distributed editing system. JViews also provides support for data persistency, collaborative editing and version control.

- *JComposer* is a direct manipulation editor for specifying and generating JViews components (itself built with BuildByWire and JViews). JComposer allows BuildByWire components and third-party Java Beans to be combined with JViews components, resulting in multi-view, multi-user, customisable componentware solutions for a wide range of editor applications.

Our toolkit is specialised from and extends the JavaBeans componentware API [JavaBeans 1996], and hence our multi-view supporting components are compatible with JavaBeans components. This means that "third party" JavaBeans components can be incorporated into JComposer environments, and JComposer environments can themselves be used as JavaBeans components. In this paper we describe our toolkit, together with an example environment constructed using it.

2. An Example Environment

Figure 2 shows an example environment constructed using JComposer. This is a simple Entity-Relationship diagrammer similar to our earlier MViewsER tool [Grundy and Venable, 1995]. It illustrates a number of features common to JComposer based environments:

- Interaction is by direct manipulation using a mouse. Users select an editing mode from the selection item e.g. Move, Resize, Hide, Add Entity, etc. They then manipulate the contents of the editing panel to build an ER model. Users have a high degree of control over the appearance of icons, including size, font style, foreground/background colours, positions of "handles" (used to resize and connect icons).

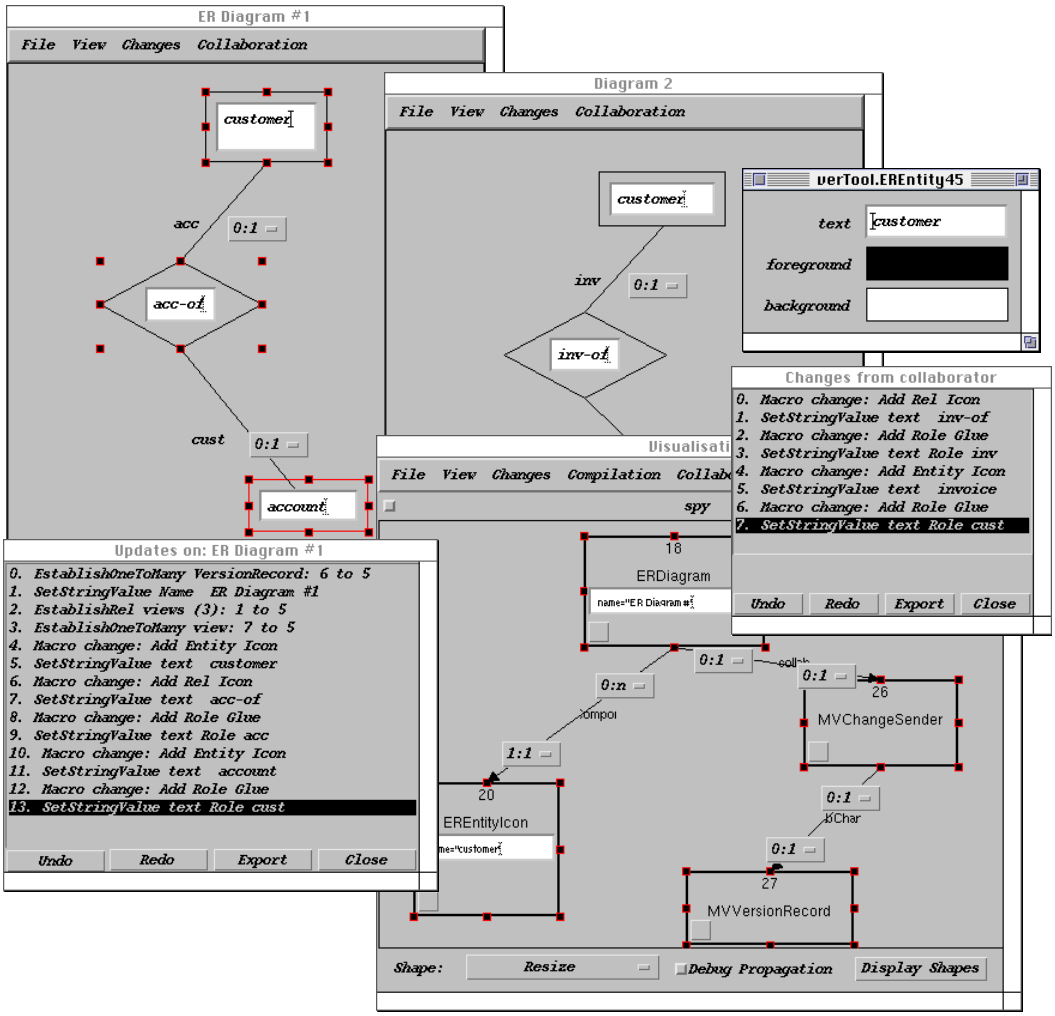

Figure 2. An example JComposer-generated Environment.

- Multiple views are supported, with overlapping information in each view. For example, the top left ER view in Figure 2 shows part of an ER model for a simple accounting system, including customers and accounts, while the top right view shows invoicing aspects. These two views are always kept consistent, so that changes to a component in one view cause corresponding changes in other views. Consistency is maintained via JViews consistency components connected to shared repository components. Descriptions of changes which can not be automatically propagated between views are shown either inside the view, indicated by highlighting icons, or can be viewed in an automatically updated changes dialog, as shown in the bottom left dialog in Figure 2.

- Runtime visualisation support is provided for all JViews components. The visualisation view in the bottom right of Figure 2 shows an entity icon component, the view layer component which contains this icon, event "listeners" which support simple notification of changes to this icon, and listener components which implement a simple collaborative editing mechanism. Environments can be both debugged and dynamically extended using visualisation views. These views provide a visualisation of JViews or BuildByWire components, which can be augmented with dynamically created JComposer components to create complex visual queries on, or event based extensions to, the environment.

- Collaboration facilities are included, based on our earlier Serendipity work [Grundy et al 96b] and provide the ability to transmit descriptions of changes from one user's environment to another. The changes can then be merged asynchronously into the other person's environment to maintain inter-view consistency. An example of changes transmitted from another user are shown in the right-hand dialogue in Figure 2.

- Support for environment persistency is provided. This currently uses a text-based serialisation mechanism, but is implemented as part of a separate component linked to the base repository. We are currently developing more sophisticated persistency mechanisms, using an extension of the JavaBeans serialisation capabilities and an object-oriented database.

In the following sections we examine each of the JComposer tools and illustrate how they can be used to construct the ER Diagrammer.

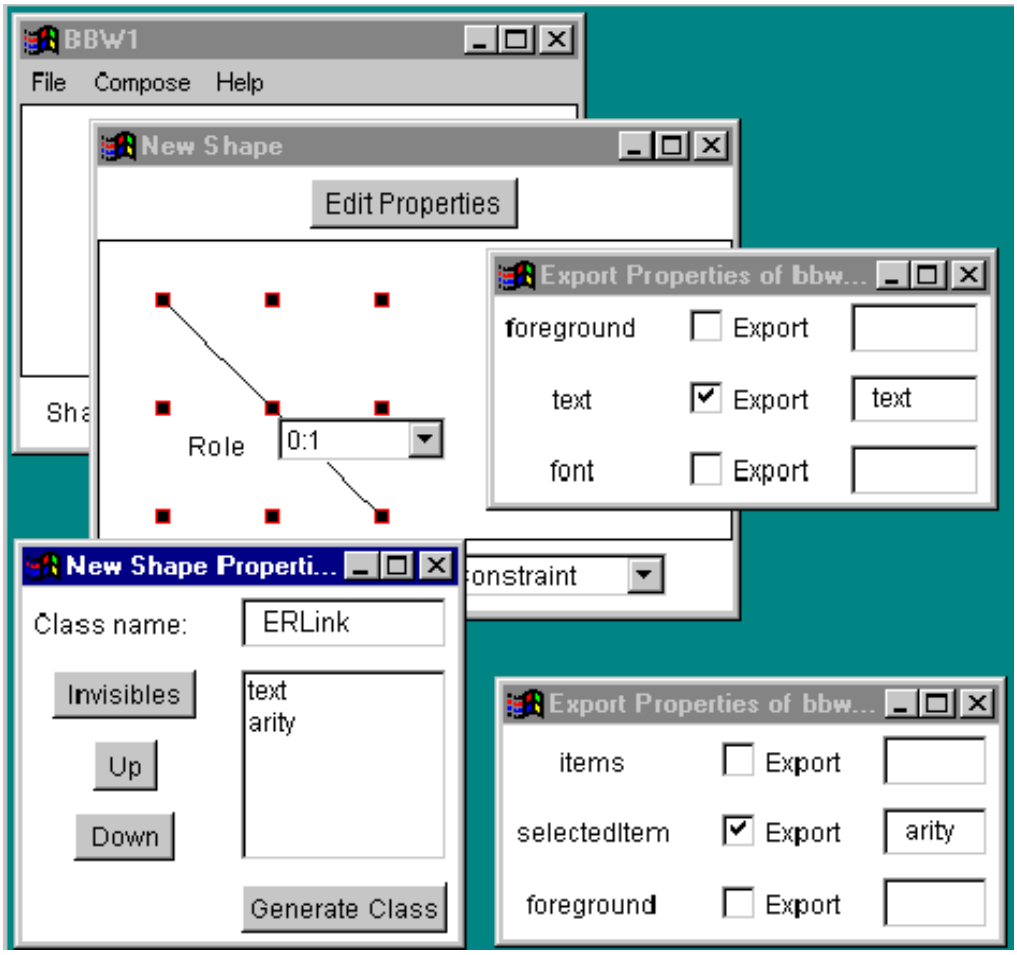

Figure 3. Composition of an ERLink Shape in BBW.

3. BuildByWire

BuildByWire (BBW) is a toolkit and direct manipulation editor for constructing constraint-based graphical user interface (GUI) components and editors for visual notations [10]. Direct manipulation is used to specify the form and interactions of new GUI components (shapes) by composing them from simple components, such as boxes, lines, icons, buttons, handles and click-regions, and container structures, such as vertical lists. "Wires" are used to compose element together by imposing constraints, such as co-location of handles, or maintenance of horizontal alignment. Constraints specify how components in a composite component behave under resizing and other direct manipulations of the composite; see [10] for further details.

Figure 3 shows BBW being used to specify a shape for the link between an entity and a relationship for the ER modeller application of Figure 1. This link includes a line, a text field for the name of the role, and a choice between 0:1,1:1,0:n,1:n for the arity of the entity in that relationship. In the final application, the link is constructed by dragging between a handle of an entity shape and a handle of a relationship shape; it is attached at each end to those handles, such that it resizes to remain attached as the end-point shapes are moved. The end-user alters the arity by selecting from the choice and alters the role through a property sheet for the link.

The form of the link is constructed in two steps (only the first step is needed for the Entity and Relationships shapes of ER Diagrammer). In the first step, a shape is composed from several existing shapes, as shown in Figure 2. The *New Shape* window shows the handles that act as the framework for the new shape, the TextShape that has been added and renamed by property sheet to "Role", and the ChoiceShape that has been customised to include the four possible values. In addition, the handles of these shapes have been hidden, to signify that they will be hidden also in the composite. The top-right of the TextShape and the top-left of the ChoiceShape have been wired to the centre handle of the framework with a EqualityMoveConstraint (which means that resizing of the composite shape results in movement of the role and arity, rather than resizing them).

Once the TextShape and ChoiceShape shapes have been added and their properties modified, it is time to specify which of their properties are to be "exported" to the composite (ie, available as properties of the composite shape). The two "export Properties.." windows of Figure 2 are used to do this. One property of each is exported, with the "selectedItem" property of the ChoiceShape exported as the property "arity" (properties which are not exported to the composite become constant values of the composite). All properties of the composite are shown in the bottom-left hand window of Figure 3, where they may be reordered; this order determines the order of properties displayed in the property sheet of an instance of the composite shape in the application. In addition to the properties from the element shapes, the designer may add so-called invisible properties: properties which have no visual form in the notational elements, but which may be changed through a property sheet.

When the composite shape has been defined, the user clicks the Generate button. BBW serialises the shapes that make up the composite and generates the source of a new class corresponding to the composite, with appropriate properties (and methods and events) and with code to make use of the serialised form of the composite elements. The source of the new class is compiled automatically, making the new shape immediately available for use.

In the second step, a new connector is constructed from the new shape, simply by selecting the shape that is to be created and attached between the two handles that have been selected by the end-user. This is based on a standard Connector class, so requires no further code generation.

To generate a new application, such as the ER Diagrammer, the designer first constructs any new shapes and connectors of the visual notation (such as the connector for ERLink and the shape for entity). Then a selection is made from the available shapes and connectors of those that are to be available for use in the new application. BBW generates Java code to automatically provide the GUI for the new application as a JavaBeans Panel.

MonoConnectors in BBW are used to define attribute icons for the ERDiagrammer. The end-user clicks on a handle of an entity and drags into open space; An attribute shape is created at the release position, with an arc connecting the attribute to the entity. A MonoConnector is defined as two shapes: the one that acts as the arc (such as a line) and the shape that is created at the mouse-release point.

Arbitrary beans may be dropped into a BeanShape. Event wires may then be used to route events from a bean, such as a button, to the exported methods of other shapes, such as to add an element to a vertical list container (as is used in one of the notational elements of JComposer itself). Hence BBW enables the user to define the visual form of their notation, as well as some of the control of the elements of the notation when they're being used by the end-user.

4. JViews

JViews is a toolkit for constructing view and repository components for multi-view systems. It includes abstractions for view and repository components, as well as inter-component relationships which are used to describe data structures, aggregate components and maintain inter- and intra-component consistency. Componentware development is supported via: a specialised form of event model to describe component state changes and general events of interest; collections of related view components being linked to one "view layer" component; related repository components being aggregated into different kinds of hierarchical repository components; and all repository components linked to one "base layer" component.

The JViews event model is similar to, but more powerful than, the JavaBeans event-listener mechanism. It utilises a "discrete change propagation and response" mechanism to provide inter- and intra-component consistency management mechanisms [Grundy and Hosking 1996a, Grundy et al 1995]. When a view or repository component is modified (i.e. its state changes) or the component generates an event of interest outside the component, a "change description" completely describing the event or state change is generated. This is propagated to the relationships the component participates in. Relationships are specialised components which can provide generic or specific event processing capabilities.They respond to received events by either: updating related components appropriately; passing the change descriptions onto related components; or ignoring them. Event receivers can be specified to "listen" before or after a change has been made to the generating component. In the before case, the receiver is notified that a change is imminent, allowing it to abort or modify the change. The after case acts similarly to the JavaBeans listener model.

This mechanism provides a homogeneous approach to the development of a wide range of consistency management mechanisms for componentware. These include:

- abstract, reusable inter-component relationships, such as. aggregation, "view-of", and constraint relationships;
- persistency, by using "persistent" forms of base layer and view layer components;
- distributed component usage, by propagating change descriptions to other users' base and view layer components for actioning;
- "adaptor" components to link JViews components to systems developed using other technologies, in particular to the JavaBeans components generated by BuildByWire;
- storing component state and event histories, to support undo/redo, version control, work coordination, etc.

The behaviour of JViews components may be modified while they are in use by specifying additional relationships and event handlers, on the fly.

Figure 4 shows part of the JViews component graph for the ER modeller in a graphical form (using the JComposer notation discussed in the next section). This describes the repository-level information for the ER modeller, including entities, roles, relationships and the repository ("base layer") component itself.

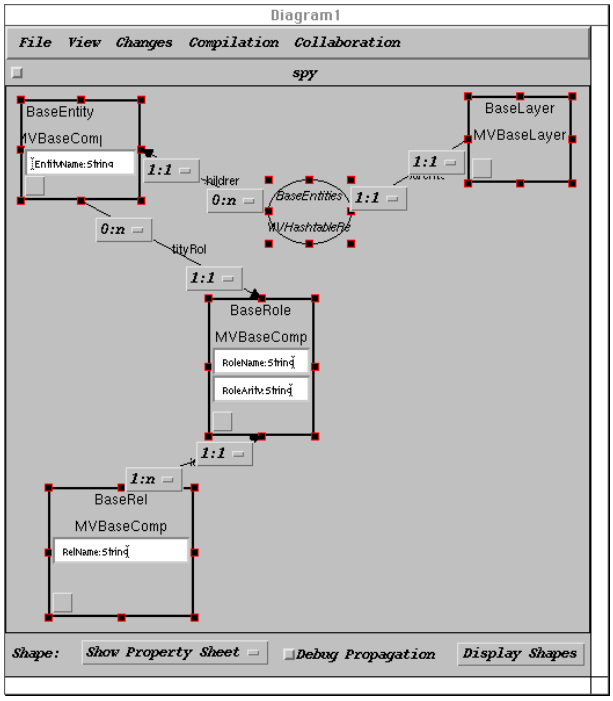

Figure 4. A JViews component structure.

5. JComposer

While it is straightforward to reuse JViews toolkit classes and link them to GUI components generated by BuildByWire, there is much repetitive housekeeping code required to do this. Figure 5, for example, shows the EREntityIcon.java class code, where BBW shape data, JViews view data and JViews repository data needs to be kept consistent. It was apparent to us that the bulk of this code could be automatically generated from a higher level description of JViews components and their interactions, leaving the developer to concentrate on less straightforward code, such as complex event handling.

```
import  java.util.*;
import  java.awt.*;
import bbw.*;
import  java.beans.*;
import  erTool.*;

public abstract class EREntityIconG extends MVViewComp
 {

  public EREntityIconG() {
    super();
    establishListeners();
  }

  public String kindName() {
    return "Entity Icon";
  }

  public abstract String userName();


  public String getText() {
    return getStringValue("text");
  }

  public void setText(String value) {
    setValue("text",value);
  }


  public EREntityIconToBase getprEntityIconToBase() {
    return (EREntityIconToBase) getOneRelated("MVViewRel",MVParents);
  }

  public void establishEREntityIconToBase(ERBaseEntity comp) {
     comp.establishEREntityIconToBase((EREntityIcon) this);
  }

  public void dissolveEREntityIconToBase(ERBaseEntity comp) {
     comp.dissolveEREntityIconToBase((EREntityIcon) this);
  }

  public ERBaseEntity getpEntityIconToBase() {
    return (ERBaseEntity) getOneRelated("MVViewRel",MVParentRelComps);
  }

  public void establishListeners() {
  }

  public MVChangeDescr beforeChange(MVChangeDescr c,
       MVComponent from, String rel_name) {
    return super.beforeChange(c,from,rel_name);
  }

  public MVChangeDescr afterChange(MVChangeDescr c,
      MVComponent from, String rel_name) {

   if(c instanceof MVSetValue) {
     if(((MVSetValue) c).getPropertyName().equals("text"))
        getEREntity().setText((getText()));
   }

    return super.afterChange(c,from,rel_name);
  }

  public MVViewRel newViewRel() {
    return new EREntityIconToBase();
  }

  public String viewRelKind() {
    return "EREntityIconToBase";
  }


  public EREntity getEREntity() {
    return (EREntity) getBBWShape();
  }



  public void propertyChange(PropertyChangeEvent evt) {
    if(hasView() && !view().processingBBWEvents &&
        view().processingBBWChange) {
      super.propertyChange(evt);
      return;
    }

      if(evt.getPropertyName().equals("text"))
        setText((getEREntity().getText()));
    super.propertyChange(evt);
  }

  public void addedBBWShape(PropertyChangeSupport2  shape) {
    super.addedBBWShape(shape);
        setText((getEREntity().getText()));
  }

  public void addedViewComp(PropertyChangeSupport2  shape) {
    super.addedViewComp(shape);
        if(getAttribute("text").equals(""))
           setText((getEREntity().getText()));
        else
           getEREntity().setText((getText()));
  }

}
```

Figure 5. An example of JViews code which can now be generated automatically.

For this reason, we developed JComposer, which allows JViews components to be specified, aggregated and related via direct manipulation of a visual notation. In a similar fashion to BuildByWire, JComposer generates appropriate specialisations of JViews toolkit classes to implement the components.

Component specification uses an object-oriented-analysis-style visual notation, specifying component specialisation, attributes, methods, and events. Figure 6 shows an example of the ER modeller structure defined using JComposer. The left-hand side components encapsulate repository data and its related behaviour. The right-hand side components represent view-layer components, which are linked to BBW shapes. The user specifies a BBW shape name in a dialogue box and the Java Beans introspection facilities are used to copy the BBW shape's properties to the JViews component. The oval icons are relationships, in this example mapping between repository and view layer components. Simple property mappings are specified for each relationship using mapping components. JComposer generates the Java classes, properties and methods to implement this ER modeller, and also generates the code to interface between BBW shapes and JViews view layer items. Third-party Java Beans can also be interfaced to JViews components, with code generated to allow JViews and the Java Beans to exhange events. Various annotations to links and relationships betwen components exist to allow JComposer users to request components be created automatically when components they are linked to are created, to specify how events from other linked components are handled, etc.

Inter-component event passing and handling behaviour is specified using a visual filter/action language, based on our Serendipity work [Grundy et al 1996b]. JViews components can "listen" to events sent to them via the relationships they have to other components. Events can be listened to not only after they have been generated by a component (e.g. after a property change) (the same as the Java Beans "listen after" approach), but also before the event has been actioned ("listen before"), before a component has received an event from another component ("handle before"), and after a component has received an event from another component ("handle after"). The later two approaches allow JViews components to intercept events being sent to other components, as well as monitor events generated by other components. JViews components may even modify the event they receive before it is passed onto other listening components, enabling very flexible event-handling schemes to be devised, and supporting the tool-based abstraction design and implementation approach for systems [Garlan et al 1992]. We have used this approach to good effect in the ViTABaL tool abstraction environment previously developed with MViews [Grundy and Hosking 1995].

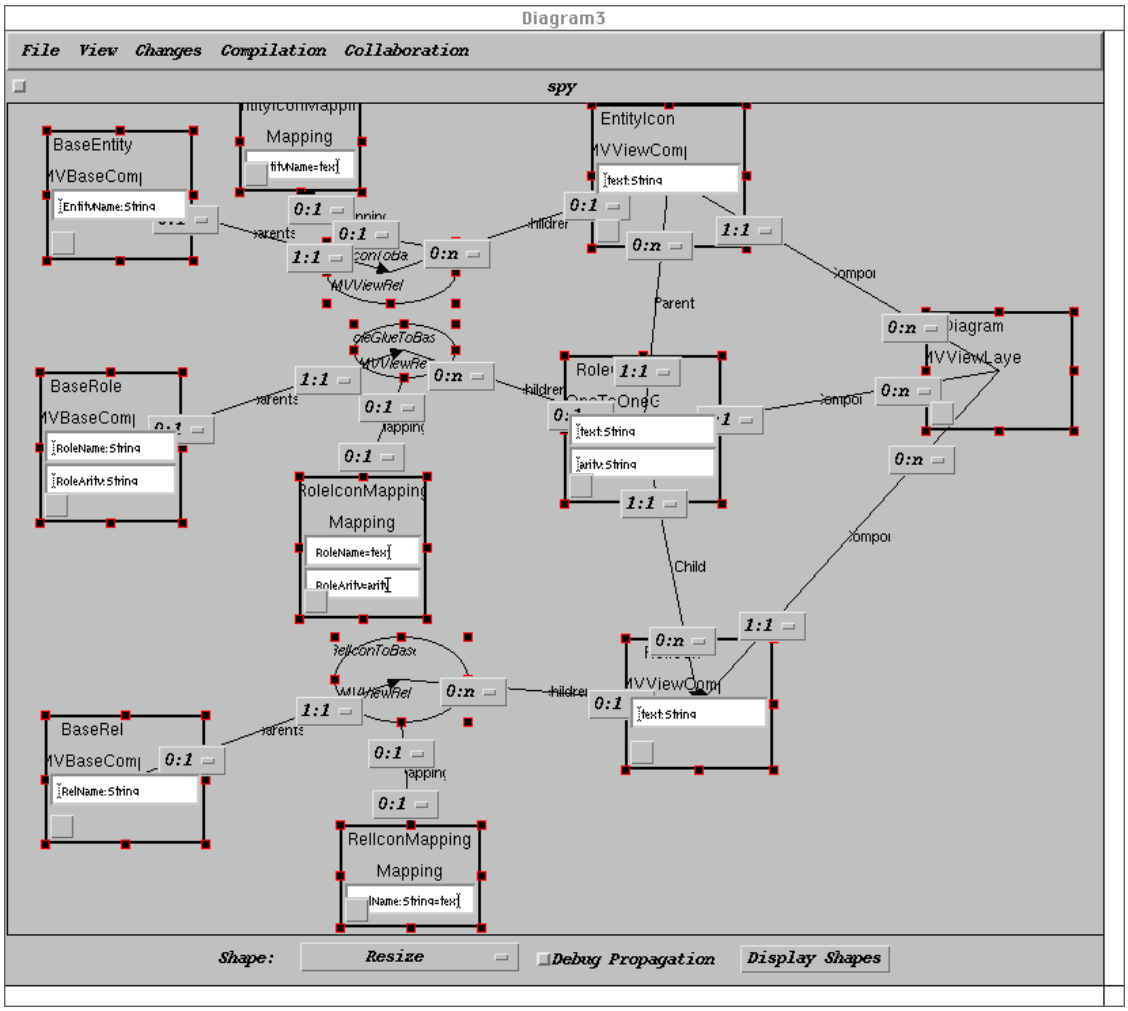

Figure 6. An example of JComposer being used to define the ER Modeller repository and views.

Any JViews component can listen to another component's events and carry out actions when these are received. Additionally, JViews uses a notion of reusable filter and action components, which unlike other components do not represent any data. Filter and action components listen to events from other components and either filter events received before passing them onto other filter/actions, or carry out some action based on the event received.

Complex filters and actions can be specified hierachically, using this visual notation, or textually, with an iconic component representation of the textual event-handling code. In the latter case, stub classes for the behaviour are generated and extended by the developer using inheritance. Subsequent regeneration recreates the stub classes, but retains the extensions, thus avoiding the information loss common in environment generation approaches. Textual extensions are currently done using a conventional text editor or Java IDE. We plan to add textual editing support to our toolkit to permit textual notation editing to be included in JComposer environments, and in particular to extend JComposer itself to support textual editing of filter/action behaviours. Figure 7 shows an example of JComposer event handling code defined heirarchically (left-hand view) and a filter defined textually (bottom text window). The hierarchical EnsureUniqueValues action listens to any relationship (in this case, the BaseEntities relationship, a hashtable), and if an entity with a name the same as an existing entity is attempted to be added, reports an error and undoes the invalid editing operation.

6. Run-time Support

Environments generated by JComposer optionally include visualisation support. This permits the state of any JViews or BuildByWire component to be observed dyamically in a visualisation view. Figure 8 shows an example of this. The Visualisation view shows a visualisation of one of the entity icons in the ER Diagram #1 view in the same figure. This extends the OOA-like notation used by JComposer to include property values for the running JViews environment. Users can expand the diagram to include other associated components and relationships, and observe their dynamic behaviour. This has obvious use when debugging environments.

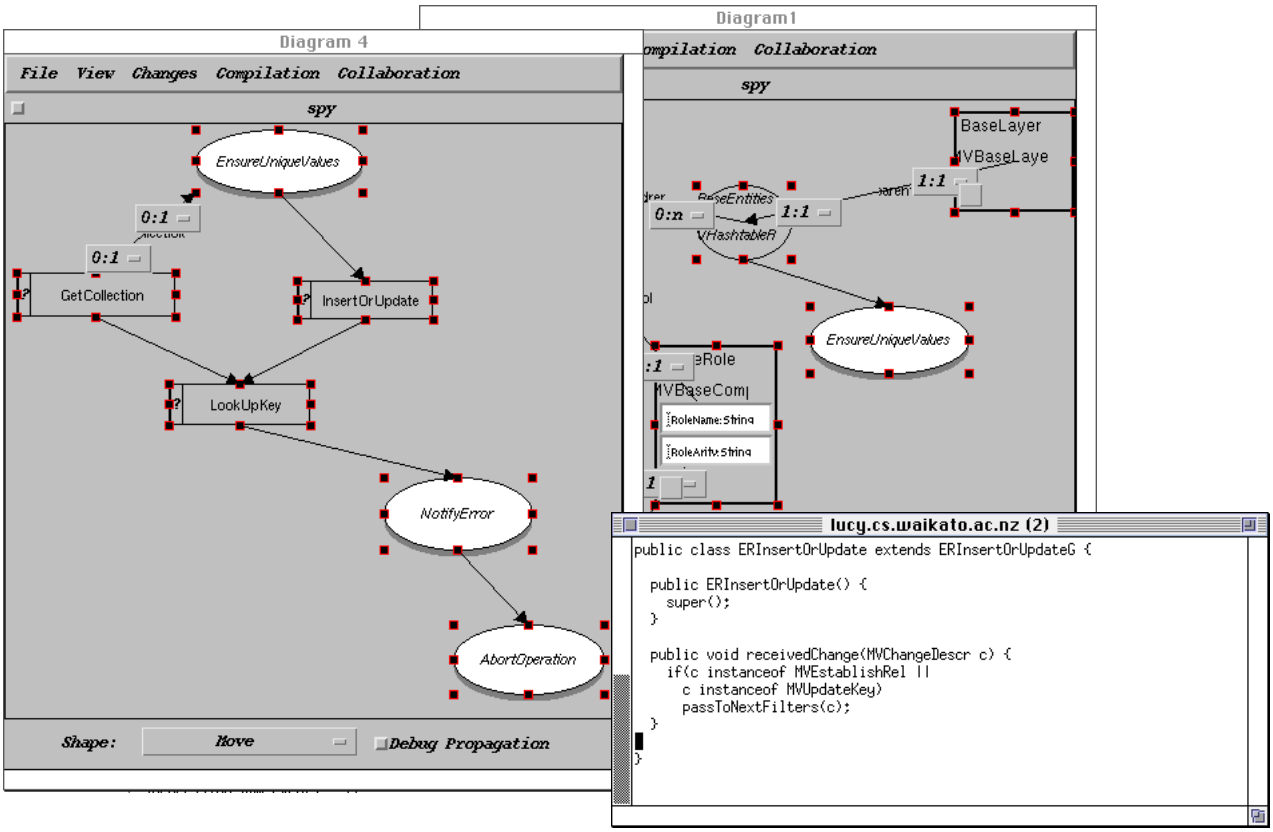

Figure 7. An example of graphical and textual event-handling behaviour specification.

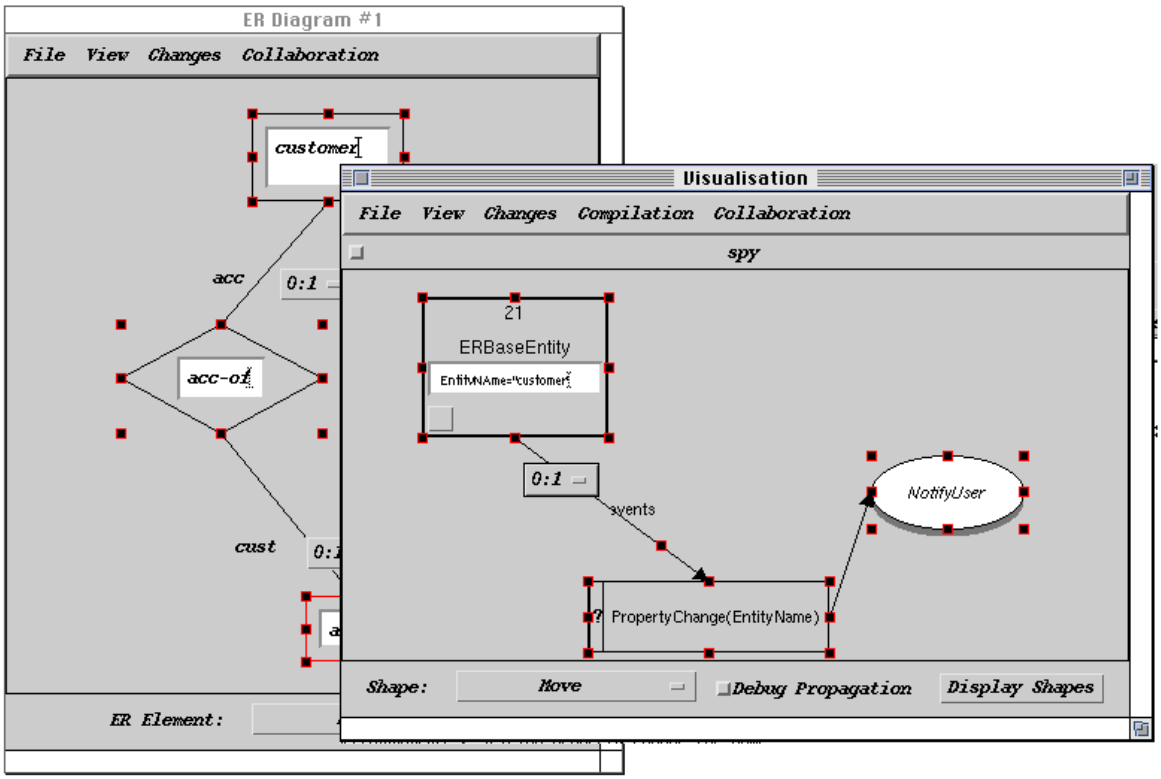

Figure 8. Example run-time visualisation and user-customisation of the JComposer ER Modeller.

In addition, JComposer specified components and relationships can be added to such diagrams. For example, a filter component and an action component have been added to the visualisation view. This causes an instance of the filter and the action to be created and relationships established between the entity icon component, the filter and the action. Changes to the entity icon component are now additionally routed to this filter/action, in this case to notify the user if the entity is renamed.

Using this process, the generated environment can have its behaviour dynamically extended. This can be used for: debugging, using the filters and actions to define visual queries; testing the addition of experimental components to the environment; end-user customisation of the environment; and with remote specification of filter/actions, for facilitating work coordination. JComposer components added in this way are limited to ones already specified and generated by JComposer. New components cannnot be generated in these views; JComposer must be used to generate them. This dynamic extension capability relies heavily on the componentware approach taken. JavaBeans reflective capabilities are used to obtain property and event handling capabilities of JComposer components. These capabilities are then used to determine appropriate ways in which the component can be dynamically related to other components.

7. Implementation

We have implemented BuildByWire and JViews in Java and using the JavaBeans API. The basic visual components and constraints in the BuildByWire editor are JavaBeans; BuildByWire generates classes for composites that are instantiated as JavaBeans. These are interfaced via adaptors to appropriate JViews view components, which are in turn related to JViews repository components via view-of relationship components. The visual filter/action specifications generate specialised forms of JViews relationships which can be established between JViews view or repository components even while in use, to add additional inter-component constraints or to handle events in different ways. JComposer allows developers to visually design and generate JViews components and inter-component relationships. The JComposer editors are themselves implemented by using BuildByWire to specify and generate the notational components and their behaviour and JViews components to provide the multiple view support. We are currently working on a specification of the JComposer system using its own notation, to allow JComposer to be incrementally extended using itself.

8. Related Work

Related work falls into two categories: systems which support component construction and systems which assist in construction or generation of multiple view environments. In the former category are tools such such as Delphi, JBuilder (for JavaBeans), and Optima (for ActiveX) [Borland 1997, Powersoft 1997]. These tools typically provide support for composition of comparatively static components, such as dialog boxes, with integrated support for textually defining event handling code. They have limited or no support for defining iconic style interaction, such as is typically required in visual notation editors, and thus have limited use in visual environment specification. Clockworks [Graham et al, 1996] supports visual programming of components, with more emphasis on specification of event handling than do the previously mentioned tools, but is still limited in the scope of tool interaction it is possible to specify. The MET++ visual environment [Wagner et al 1996] supports the composition of object-oriented framework components for multimedia applications. The MET++ environment only supports simple component connectivity using mappings between wrapped MET++ framework instances, and does not support the range of structural and event-handling mechanisms of JComposer. While MET++ components can be combined with this environment, new iconic representations of the style supported by BuildByWire can not be directly produced.

In the latter category is Escalante and its associated GrandView generation tool [McWhirter and Nutt 1994], which provides very good assistance for visual notation and environment specification, together with support for multiple views. The latter is, however, limited in comparison to JViews and the JComposer visual filter/action language. More importantly, Escalante does not take a componentware approach, thus making it difficult to incorporate third party components, or to embed generated environments as components in other applications.

9. Summary and Future Work

We have been developing componentware solutions for the construction of multi-view, component-based software. BuildByWire provides a direct manipulation approach to the development of reusable UI components and entire componentware editors. JViews provides a toolkit for the development of view and repository components, including flexible inter- and intra-consistency management mechanisms. JComposer allows developers to better design, generate and inter-relate (“wire”) JViews multiview components than would be possible with a tool based purely on JavaBeans. We have built a multi-view, multi-user Entity-Relationship modeller using JComposer.

We are currently refining and extending JComposer. An initial goal is to complete the JComposer self specification, to allow JComposer to be extended using itself. In addition, we are working on extensions to provide textual view support, and textual-visual consistency, as we developed in our MViews work [ref]. This involves issues of the most appropriate way to specify textual components and textual editing.

In parallel, we also plan to use JComposer to generate new environments. Initially this will replicate some of our MViews-based environments, although using a componentware. For example, we plan to generate an object-oriented programming environment, similar to SPE [Grundy et al 1995], but based on the Unified Modelling Language [ref] OO analysis notation for visual aspects. Other environments planned include CSCW tools, programming environments, and HCI and SE modelling tools.

More into the future, we have an interest in the generation of event based environments using alternative types of user interface components, such as VRML nodes [ref] wrapped as Java beans [ref to VRML beans proposal].

References

[Active X 1996]
"ActiveX", by Microsoft, http://www.microsoft.com/activex/.

[Amor et al 1995]
Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., Grundy, J.C., Directions in Modelling Environments, *Automation in Construction*, Vol. 4 , Elsevier Science Publishers, 1995, 173-187.

[Borland 1997]
JBuilder Java application development environment, Borland International Inc, 1997.

[Garlan et al 1992]
Garlan, D., Kaiser, G.E., and Notkin, D., "Using Tool Abstraction to Compose Systems," *COMPUTER*, vol. 25, no. 6, 30-38, June 1992.

[Graham et al 1996]
Graham, T.C.N., Morton, C.A., and Urnes, T., "ClockWorks: Visual Programming of Component-Based Software Architecture," *Journal of Visual Languages and Computing*, 175-19, July 1996

[Grundy and Hosking 1995]
Grundy, J.C. and Hosking, J.G. ViTABaL: A Visual Language Supporting Design by Tool Abstraction, In *Proceedings of the 1995 IEEE Symposium on Visual Languages,* Germany, 1995, IEEE CS Press, pp. 53-60.

[Grundy and Hosking 1996]
Grundy, J.C., and Hosking, J.G., Constructing Integrated Software Development Environments with MViews, *International Journal of Applied Software Technology,* International Academic Publishing Company, Vol. 2, No. 3/4, 1996.

[Grundy et al 1995]
Grundy, J.C., Hosking, J.G., Fenwick, S., Mugridge, W.B., Connecting the pieces, Chapter 11 in *Visual Object-oriented Programming*, M. Burnett, A. Goldberg, T. Lewis Eds, Manning/Prentice-Hall, 1994.

[Grundy et al 1996a]
Grundy, J.C., and Hosking, J.G., Mugridge, W.B., Supporting flexible consistency management via discrete change description propagation, *Software - Practice and Experience*, Vol. 26, No. 9, September 1996, Wiley, 1053-1083.

[Grundy et al 1996b]
Grundy, J.C., Hosking, J.G., Mugridge, W.B. Low-level and High-level CSCW support in the Serendipity process modelling environment, In *Procedings of OZCHI'96*, Hamilton, New Zealand, November 24-27, 1996, IEEE CS Press, pp.69-76.

[Grundy and Venable 1995]
Grundy, J.C., Venable, J. Providing Integrated Support for Multiple Development Notations, in *Proceedings of CAiSE '95*, Finland, June 1995, Lecture Notes in Computer Science 932, Springer-Verlag, pp. 255-268.

[Hosking et al 95]
Hosking, J.G., Fenwick, S., Mugridge, W.B., Grundy, J.C., Cover yourself with Skin, in *Proceedings of OZCHI'95*, Wollongon, Australia, Nov 28-30, 1995, pp. 101-106.

[JavaBeans 1996]
"JAVABEANS™ API", by JavaSoft, http://www.javasoft.com:80/products/apiOverview.html

[McWhirter and Nutt 1994]
McWhirter, J.D. and Nutt, G.J., "Escalante: An Environment for the Rapid Construction of Visual Language Applications," in *Proceedings of the 1994 IEEE Symposium on Visual Languages,* IEEE CS Press, 1994.

[Mugridge et al 1996]
Mugridge, W.B., Hosking, J.C., Grundy, J.C. Towards a Constructor Kit for Visual Notations, In *Proceedings of OZCHI'96*, Hamilton, New Zealand, November 24-27, 1996, pp. 169-176.

[Opendoc 1996]
"OpenDoc Programmer's Guide", by Apple Computer, Inc, Addison-Wesley, 1996.

[Powersoft 1997]
Optima++ RAD tool, Powersoft Business Group, Sybase Inc, Canada, 1997.

[Wagner et al 1996]
Wagner, B., Sluijmers, I., Eichelberg, D. and Ackerman, P. Black-box Reuse within Frameworks based on Visual Programming, In Proceedings of The First Component Users Conference, Munich, July 15-19 1996.