JComposer/JViews Tutorial

John Grundy

December 2000

This document provides a brief tutorial to using JComposer and JViews to build multi-view editing tools. I recommend you read the following papers to gain an idea about how JViews and JComposer work, from a conceptual level at least:

> Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, Journal of Information and Software Technology: Special Issue on Constructing Software Engineering Tools, Vol. 42, No. 2, January 2000, pp. 117-128.
>
> Grundy, J.C., and Hosking, J.G., Mugridge, W.B**.,** Visual Specification of Multi-View Visual Environments, In Proceedings of the 1998 IEEE Symposium on Visual Languages, Halifax, Canada, Sept 4-7, IEEE CS Press, pp. 236-243.

All the usual caveats apply – this is research software that in many places has been developed to proof-of-concept level and no more. Don't attempt to use it for anything other than exploratory research development.

Note that in the following tutorial we drawn upon completed parts of the JComposer/JViews systems. We don't use BuiltByWire's code generation tool as that is incomplete, and nor do we use filter/actions in JComposer itself, as code generation from these is incomplete.

The main steps in this process are summarised below:

1. Define your tool's meta-model using JComposer
2. Implement your tool's icons/glue using BuiltByWire's framework
3. Define your tool's icon/view component mappings using JComposer
4. Define view/base component mappings using JComposer
5. Generate JViews classes from JComposer
6. Implement view component mapping functions in Java
7. Implement constraints (view & base) in Java
8. Implement code generation, reverse engineering, tool integration etc in Java

I use a simple ER modeller application as the example application to build in this tutorial. The ER modeller needs to provide the following functionality:

- Entities – with a name. Each entity has one or more icons in one or more views.
- Attributes – in this example, defined as text connected to an entity icon
- Relationships – these connect entities and have an arity and role name.

1. Defining the ER meta-model in JComposer

The meta-model elements for a CASE tool typically include:

- A "Base Layer", which organises the set of model elements
- One or more collections of base components of various types e.g. entities
- Various "Base Components" which represent model elements
- Inter-relationships between model elements

For our ER modeller, we'll want:

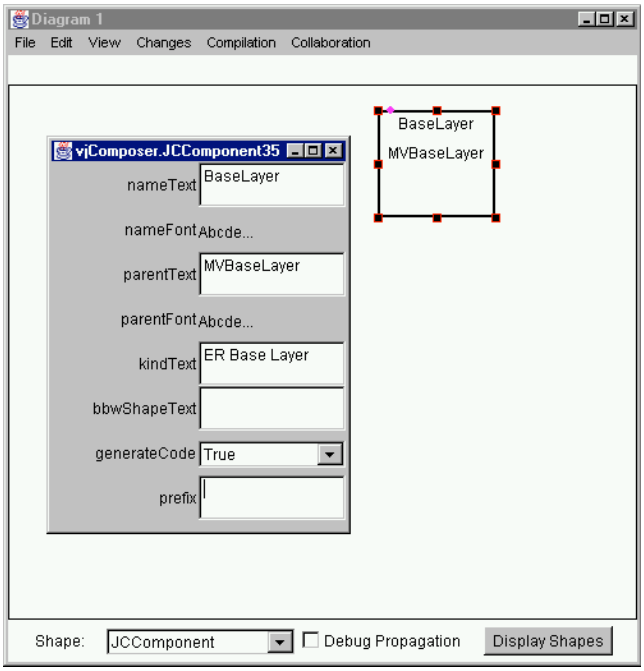
- A base layer (derived from the JViews MVBaseLayer component)
- A collection of entities (derived from the JViews MVHashtableRel relationship component)
- Base entities (derived from MVBaseComp)
- Base relationships (derived from MVBaseComp)
- Base attributes (derived from MVBaseComp).

Each base entity will be linked to zero or more relationships and zero or more attributes.

1.1 Start JComposer and create a new project and "base layer". Click on "New" on the first dialogue that opens and then "New" on the second. Save this project by selecting "Save Project" from the "File" menu of the window named "Diagram 1"

1.2 Create a "JCComponent" shape by clicking & draging in the view. Type in "BaseLayer"<return> for its NameText and "MVBaseLayer"<return> for its ParentText. Select "True" for generateCode.

The following diagram shows the definition of the base layer in JComposer:

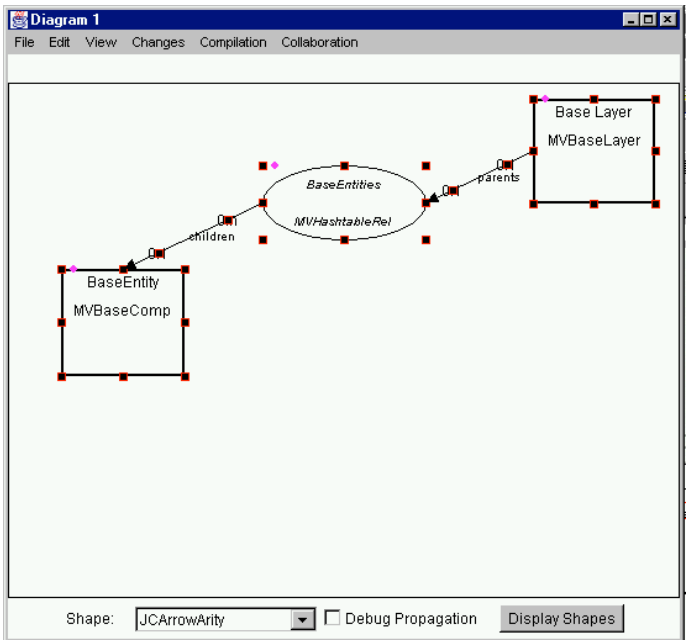


1.3. Add a hashtable relationship to track base entities. Select the JCRelnShape from the Shape menu and click & drag to add. Set nameText="BaseEntities" and parentText="MVHashtableRel". Set generatecode to True.

1.4. Connect the base layer component to the hashtable it will own. Select "JCArrowArity". Click on one of the handles on the Base layer component and drag to and release on one of the hashtable relationship handles. Name the relationship nameText="parents". DON'T SET GENERATECODE TO TRUE! DON'T GIVE THE RELATIONSHIP A DIFFERENT NAME!! This is because the "parents" and "children" relationships are defined in the superclass - we don't want Jcomposer to generate code to manage this relationship as our BaseEntities relationship component inherits this functionality anyway.

1.5. Create a base component "BaseEntities", whose parent is "MVBaseComp". Connect from BaseEntities to BaseEntity, calling this relationship "children".

Your model should look vaguely like:



Let's have a pause and check things are defined more-or-less correctly. We'll generate the JViews classes that will implement this part of the tool meta-model.

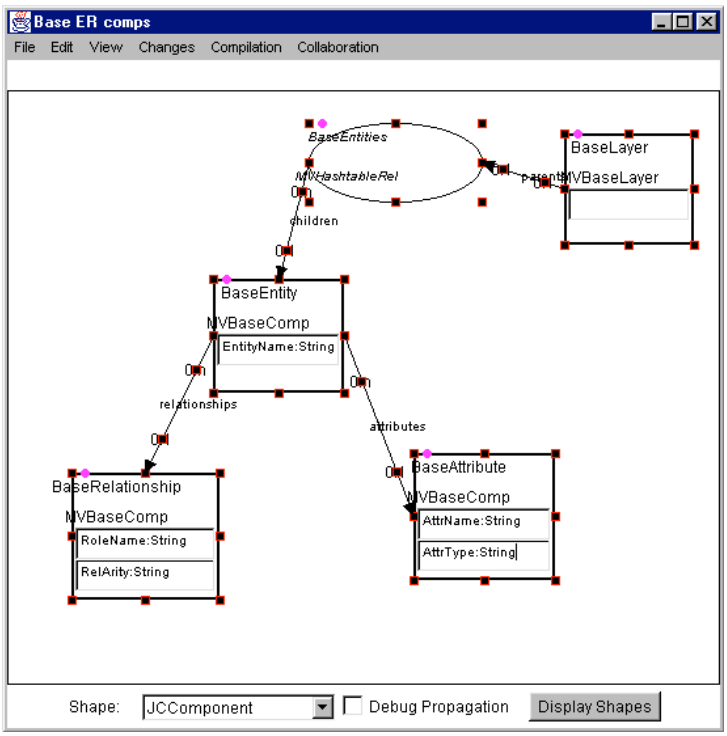
1.6 Go to "Compilation" and select "Set Package Info". Set the prefix to "ER" and the package to "ERModeller". Create a directory under the JComposer directory called "ERModeller". Select "Generate Code" and specify the ERModeller directory as output (give any name to the save file e.g. "er").

Check there are files called ERBaseEntity.java, ERBaseEntityG.java, ERBaseEntitiesG.java etc. (should be six of them at this stage). Check they compile by going javac *.java in ERModeller directory. You need JComposer and BBW directories in your class path (but these should be there if JComposer is working!).

1.7. Add BaseRelationship (parentText="MVBaseComp") and BaseAttribute (parentText="MVBaseComp") components. Connect these to BaseEntity by a JCArrowArity relationship. Call one "relationships" and one "attributes". Set the generateCode to True, and choicefrom to "0:n".

Remember to periodically save your project. I recommend periodically backing it up too!!

1.8 Add attributes to components. Right-click on BaseEntity and select "Add element". Type in "EntityName:String". Add an attribute to BaseRelationship "RoleName:String" and another "RelArity:String". Add two to BaseAttribute, "AtrrName:String" and "AttrType:String". Your model should look like the following:



Regenerate the JViews classes. The tool model implementation is now complete, apart from constraints etc which you can program in Java by adding code to the classes without the "G". You can modify the meta-model in JComposer and regenerate it, in which case the "G" suffixed classes are replaced. There is not currently a reverse engineering facility in JComposer, so make changes in it, not to the "G" classes directly!

2. Implement your tool's icons/glue using BuildByWire's framework

Sadly the BBW interactive never quite got finished to enabling us to do this in a general sense, so we'll implement these in Java via copy-and-paste from other tool implementations.

2.1. Copy the files from my BBW.ERModeller directory that comes with this tutorial. These implement a BBW panel (drawing area), shapes (EREntityShape, ERRelationshipLink), and events (ERNewEntityShape, ERNewRelationshipLink). Have a look over these – most are straightforward (except the panel, but even that isn't too bad). You can try out the ER modeller editor by running the ERM programme in the BBW directory i.e. go java ERM to run. Right click on icons and select Properites to set e.g. its name value.

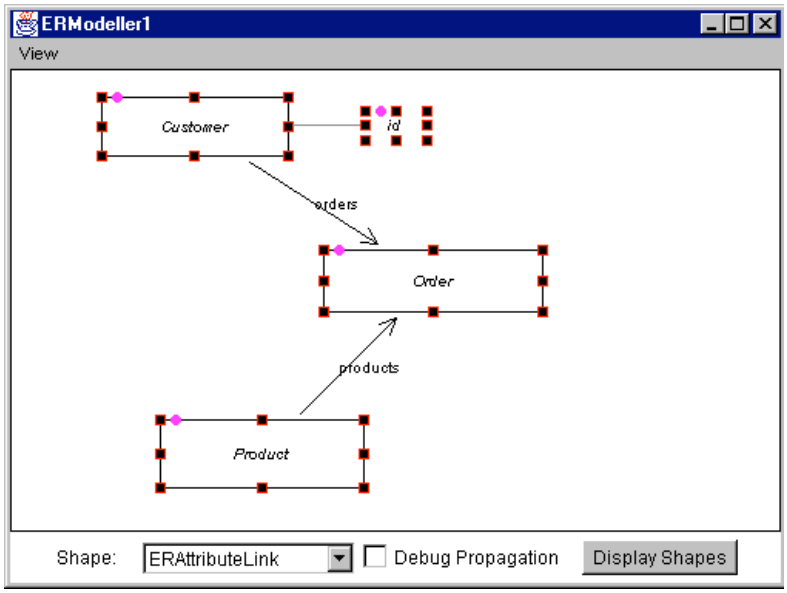
We will create a new ERAttributeShape and ERAttributeEvent and modify the ERPanel to allow attribute shapes to be added to it.

2.2. Look at the EREntityShape.java file. This uses a BBW "TextShape", which provides a line of text as a component shape. The Entity Shape extends this to provide a simple border around the text. Composite shapes with e.g. multiple text lines, multiple shapes etc can also be defined in BBW (see the various JComposer tool shape examples in the BBW.jComposer folder). Copy this into a ERAttributeShape.java file. Modify this so "Entity" is changed to "Attribute". Change the paintBackground() method so it no longer paints a rectangle (remove the g.drawRectangle() call).

2.3. Copy the ERNewEntityEvent.java file into ERNewAttributeEvent.java and modify similarly. Edit the ERPanel.java file and copy the function newEntity() and name the copy newAttribute(). Change "Entity" into "Attribute" in this new function. Add ERAttributeShape to the tools string at the boom of the ERPanel.java file.

You can now try out the attribute shape by compiling the files and running the ERM application again. An ERAttributeShape should appear in the drop down Shape menu and be able to be added to the view. Right click on the icon and select Properites to set its name value.

2.4. We'll define a connector between ERAttributeShape and EREntityShape objects. Copy the ERRelationshipLink.java and ERNewRelationshipLinkEvent.java files to ERAttributeLink.java and ERNewAttributeLinkEvent.java files. Rename "Relationship" to "Attribute" in these files. Change the paint() method so it draws e.g. Color.Gray lines with no arrows. Edit Erpanel.java and add a line addConnectorTool("ERAttributeLink", "ERAttributeLink"); below the existing one for adding relationship links. Run the tool again and check you can add attribute links between icons. An example of it running is shown below:

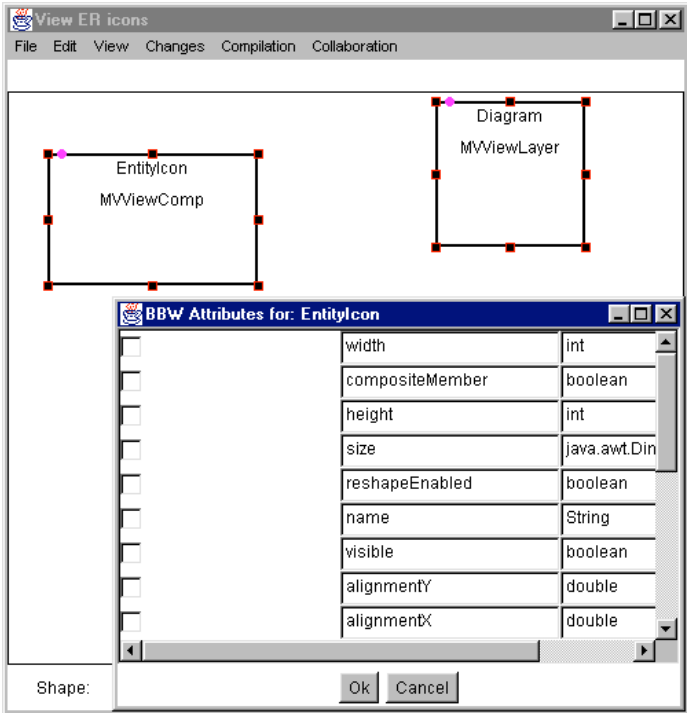


3. Define your tool's icon/view component mappings using Jcomposer

We'll now define some view layer components and Jcomposer that will use the above BBW components to display and edit ER models.

3.1 Start up Jcomposer (if not still running) and load your last saved ER modeller meta-model specification file. Create a new view. You can give it a name if you want (select Rename View from the View menu). Add a component to this view and call it "Diagram" with parentText="MVViewLayer". Set the BBWShape property to "ERModeller.ERPanel". Set generateCode=True. This tells Jcomposer to generate a whole bunch of code to detect ERPanel events, create ERPanel icons etc etc. You can regenerate the ER modeller component .java files if you like, and have a look at the code generated in the ERDiagramG.java file.

3.2. Add a component "EntityIcon", parentText="MVViewComp", generateCode="True" and BBWShape = "ERModeller.EREntityShape". Right-click on this and select Get BBW Attributes. If nothing happens, ensure you spelt the BBWShape name right (and check the command line for an error from Jcomposer). This will bring up a dialogue like that below:

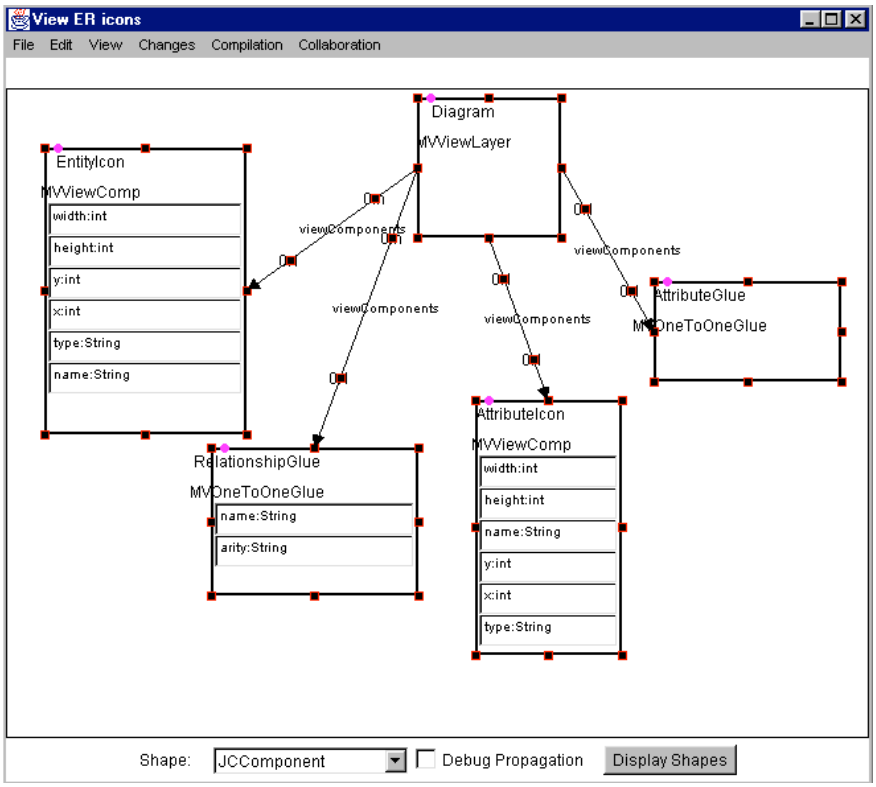


Select width, height, x, y, name and type. Select Ok. You'll see a bunch of attributes added to the Entity Icon component. This has told Jcomposer to keep these attributes of the BBW JavaBean and the Jcomposer View Component consistent when either changes.

Do the same as above for the AttributeShape i.e. define a Jcomposer AttributeIcon, upload its attributes from the BBW ERAttributeShape bean etc.

3.3. Define a Jcomposer component "RelationshipGlue", parentText="MVOneToOneGlue", generateText="True" and BBWShape="ERModeller.ERRelationshipLink". Upload the BBW shape attributes, but only select name and arity, NOT x, y, width and height (as these are recomputed from the connected icon positions). Repeat to create a AttributeGlue Jcomposer component (BBWShape="ERModeller.ERSAttributeLink).

3.4. Link the ERDiagram component to each newly defined view component icon, using ArrowArity. Set the name to "viewComponents" and generateCode=False (its an inherited relationship defined in MVViewLayer and MVViewComp). Your meta-model for the ER modeller view components should look like:



At this stage we can try out the partially-complete modeller. I've provided the classes ERApplication.java (which contains the main() function) and ERProject.java (which groups multiple ER models open at one time).

3.5. We need to make one small addition to the ERBaseLayer.java file. Open this and add the following function definition:

```
public void initialise()
  {
   // should have a "create linked" do this!

   ERBaseEntities be = new ERBaseEntities();
   be.init("ERBaseEntities",this);
   be.setHandleAfterRel("ERBaseEntities");

   ERDiagram er_view = new ERDiagram(this,nextViewName());
   er_view.show(); // show view's frame...

  }
```

This tells the ER modeller base view to initialise the base entities hashtable relationship and to create a default view.

Compile all of the files in Jcomposer.ERModeller. To run the ER modeller go:

```
java ERModeller.ERApplication
```

Try out editing the ER modelling view. All of the icons and glue, when added, should open a property dialog. This indicates they are correctly linked to Jcomposer view components. Try saving and reloading an ER model. If it returns to its former state, everything is good!!

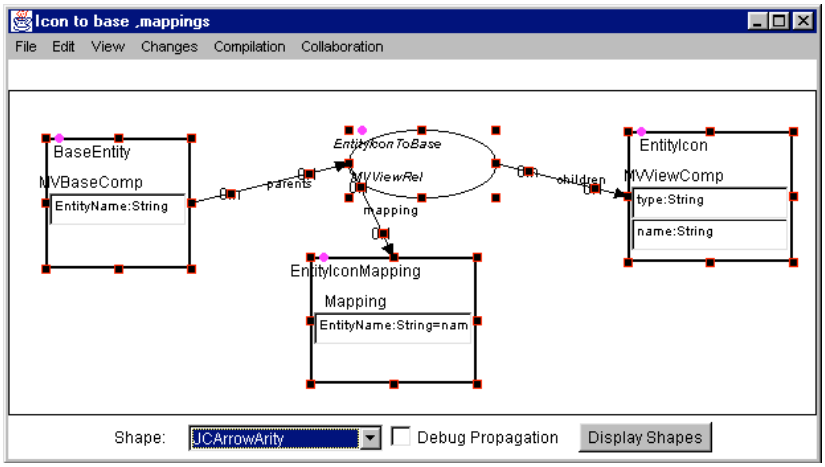
4. Define view/base component mappings using JComposer

There is one more thing Jcomposer can do before we have to resort to Java programming (for view component->base component mappings, constraint enforcement and any other facilities like code generation etc).

4.1. Create a new Jcomposer view. Add components "BaseEntity" and "EntityIcon". Their parent class names should appear. Right-click on the component icons and select "Show Attributes". Check them all and then click Ok.

We will now specify a mapping from view component attributes to base component attributes and get Jcomposer to generate code to keep these synchronised.

4.2. Create a new JCRelnShape (relationship) and call it "EntityIconToBase", parentText="MVViewRel", generateCode="True". Connect from the BaseEntity to the EntityIconToBase relationship with JCArrowArity, calling this "parents", generateCode=False. Connect from the EntityIconToBase relationship to EntityIcon component, calling this relationship "children", generateCode=False. Create another component, call it "EntityIconMapping", parentText="Mapping", generateCode=False. Connect from the EntityIconToBase to EntityIconMapping using JCArrowArity, calling this "mapping", generateCode=False. Add an element to the EntityIconMapping component (Right-click on icon, select Add Element), naming it "EntityName:String=name". This specifies the BaseEntity.name attribute is synchronised with the EntityIcon.name attribute. The view should look something like:



Repeat this process for the RelationshipGlue<->BaseRelationship and AttributeIcon<->BaseAttribute components.

5. Implement view component mapping functions in Java

We now have to resort to Java programming to complete the ER modeller. There are many features that Jcomposer could be enhanced with, but most either never quite made it to the top of the priority list, or those that are there (like constraint generation, automated component creation and initialisation etc) never quite got finished so that they are robust enough for general use…

We will firstly use Java to implement "mapping functions" in the Jcomposer view icon and glue components. These functions simply locate a base component to map a view component to, usually after the view component has been created or a property has been set.

5.1. Open the EREntityIcon.java file. Add the following function:

```
public MVBaseComp mapComponent(boolean do_map) {
```

```
System.out.println("in mapComponent()");

      if(getName().equals(""))
         return null;

System.out.println("trying to map to base...");
System.out.println("getName() = '"+getName()+"'");

      ERBaseLayer base_layer = (JCBaseLayer) view().getBaseLayer();
      ERBaseEntity base_comp = base_layer.findBaseEntity(getName());

      if(do_map) {
         if(base_comp != null) {
            mapToBase(base_comp);
            return base_comp;
         } else {
            base_comp = new ERBaseEntity(base_layer);
            mapToCreatedBase(base_comp);
            base_comp.init(base_layer);
            base_layer.establishBaseEntities(base_comp);
            return base_comp;
         }
      }
         else return base_comp;

  }

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

         if(c instanceof MVSetValue) {
            if(((MVSetValue) c).getPropertyName().equals("name") &&
               baseComp() == null && hasView()) {
System.out.println("afterUpdate for EntityIcon calling mapComponent...");
                  mapComponent(true);
            }

         }

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

This function implements a simple mapping where when the user supplies the name of the entity, it looks up the corresponding base entity and “maps” itself to it. The functions … perform various Jviews component initialisation e.g. constructing a view relationship, initialising this and linking the base and view components as parent and child respectively via the view relationship.

You’ll also want to add a small look-up function to ERBaseLayer.java:

And change the “userName()” function in ERBaseEntity.java:

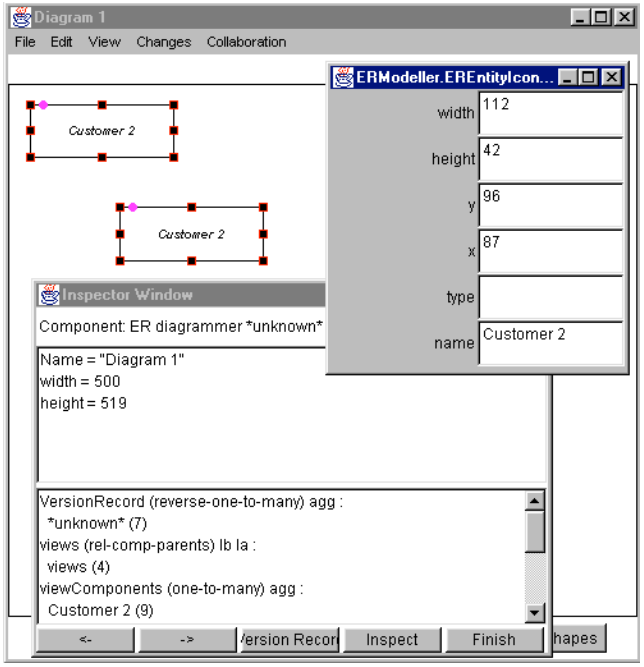
```
 public String userName() {
```

```
  return getEntityName();
 }
```

This is used by the ERBaseEntities hashtable as the “key” for base entity look-up.

Recompile and run your ER modeller. Add an entity icon and call it “Customer”. Check the command line to see the mapping function is being called etc. You can also see the state of components using the object inspector. Select “Inspect View” from the View menu. This will show the attribute values and relationships of the view. Look in the relationships list to see the components linked to the view component via viewComponents. Select the Customer component and click on Inspect. This will show the values of its attributes/relationships. Look at the MVViewRel relationship (if it is there!). There should be one item - select and click Inspect. This should have one parent and one children. Select the parent and click Inspect - this is the base entity component created by the mapping function.

Add another called “Customer”. Rename this and check both entity icon names change. If they do, your base and view components are being synchronised by Jviews!



6. Implement constraints (view & base) in Java

Another way to debug Jviews applications is via the change propagation mechanism between Jviews components, used to manage a whole range of things. As an example, add the following function to the SEBaseEntities.java class:

```
public MVChangeDescr afterChange(MVChangeDescr c, MVComponent from, String rel)
 {
 System.out.println(“Done a relationship change: “+c);

  return super.afterChange(c,from,rel);
 }
```

Running the ER modeller now should print out events e.g. EstablishRel when entities are created and named.

Note however that when an entity is renamed, its key in the hashtable is not updated! We can use Jviews' change propagation to have the BaseEntities hashtable relationship told about the rename of an entity, and even have it reject renaming an entity to a name that is already used (or have it annotate the new name to indicate this).

To do this we need to have the BaseEntities relationship component told when one of the base entities it manages is about to be renamed and has been renamed. We can programmatically do this by adding the following function to ERBaseEntities.java:

```
public void establish(MVComponent parent, MVComponent child)
  {
   super.establish(parent,child);
   if(child != null) {
      child.setListenBeforeRel(relName());
      child.setListenAfterRel(relName());
   }
  }
```

Alternatively, we can get Jcomposer to generate this code in the generated ERBaseEntitiesG.java file. In Jcomposer, right-click on the relationship between BaseEntities and BaseEntity and set parentListenBefore and parentListenAfter to True. Then regenerate the ER modeller files. This ensures the BaseEntities relationship informed to all changes being made to all of its base entity components.

We can implement functions to e.g. abort an invalid rename, simply change the name back or annotate it to make it still unique. The following code does the last one:

```
MVChangeDescr lastChange = null;
  String lastUserName = null;
  MVComponent lastFrom = null;

  public MVChangeDescr beforeChange(MVChangeDescr c, MVComponent from, String rel)
  {
   if(lastChange == null && isChild(from)) {
      lastChange = c;
      lastUserName = from.userName();
      lastFrom = from;
   }

   return c;
  }

  public MVChangeDescr afterChange(MVChangeDescr c, MVComponent from, String rel)
  {
   if(c == lastChange && from == lastFrom) {
      // send changes to base layer (should gen. code for this...)
      Enumeration e = parents();
//       while(e.hasMoreElements()) {
//          ((ERBaseLayer) e.nextElement()).broadcastAfter(c);
//       }

System.out.println("BaseEntities::afterchange got "+c);
      if(!from.userName().equals(lastUserName)) {
System.out.println("BaseEntities checking for unique new ID...");
         // change of key for base entity
         // need to check still unique!
```

```
        if(get(from.userName()) != null) {
          //
           // tag with " (#)" annotation to ensure unique
           // user can then rename...
          //
          // could reject change by throwing an exception or reversing the
          // change made to the base entity
          //
           int next = getIntValue("next")+1;
           setValue("next",next);
           ((ERBaseEntity) from).setEntityName(from.userName()+" ("+next+")");
           changeKey(lastUserName,from.userName(),from);
        } else
          // change OK so get hashtable to update key
          // generates ChangeKey event...
           changeKey(lastUserName,from.userName(),from);
      }
      lastChange = null;
      lastUserName = null;
      lastFrom = null;
    }

    return c;
  }
```

7. **Implement code generation, reverse engineering, tool integration etc in Java**

These are exercises left to the reader… ☺ Have a look at the implementation of Jcomposer itself for an example of code generation (see JCBaseComp.java). Some examples of tool integration exist in Serendipity-II, via some of its plug-in action components…

For the ER modeller, we could e.g. generate RDBMS CREATE TABLE commands (code generation), import RDBMS table schema (reverse engineering), run CREATE TABLE etc commands on a database e.g. via JDBC (tool integration) and so on…

References

Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, Information and Software Technology Vol 42, No. 2, Special Issue on Constructing Software Engineering Tools, Elsevier Science Publishers.

Grundy, J.C. A method and environment for distributed component engineering, In Proceedings of the 2000 Conference on Software - Methods & Tools, Wollongong, Australia, Nov 6-10, 2000, IEEEE CS Press.

Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, In proceedings of the 2000 IEEE Symposium on Visual Languages, Seattle, Washington, Sept. 14-18 2000.

Grundy, J.C. and Hosking, J.G. Developing Adaptable User Interfaces for Component-based Systems, In Proceedings of the 1st Australasian User Interface Conference, Canberra, Australia Jan 30-Feb 2 2000, IEEE CS Press, pp. 17-25.

Grundy, J.C. Engineering component-based, user-configurable collaborative editing systems, Engineering for Human-Computer Interaction, Chatty, S. and Dewan, P. Eds, February 1999, Kluwer Academic Publishers.

Grundy, J.C. and Hosking, J.G. Human-Computer Interaction Issues for the Configuration of Component-based Software Systems, In Proceedings of OZCHI'99, Wagga Wagga, Australia, Nov 1999, pp. 37-43.

Grundy, J.C., Hosking, J.G., Mugridge, W.B., Apperley, M.D. A decentralised architecture for software process modelling and enactment, IEEE Internet Computing: Special Issue on Software Engineering via the Internet, Vol. 2, No. 5, September/October 1998, IEEE CS Press, pp. 53-62.

Grundy, J.C. and Hosking, J.G. Serendipity: integrated environment support for process modelling, enactment and work coordination, Automated Software Engineering: Special Issue on Process Technology, Vol. 5, No. 1, January 1998, Kluwer Academic Publishers, pp. 27-60.