DiaGen

The Component Tutorial

http://www2.informatik.uni-erlangen.de/DiaGen/
diagen@immd2.informatik.uni-erlangen.de

February 29, 2000

Contents

1 Aspects of a diagram component 4

2 Component skeletons 5

2.1 Generating a component skeleton 5

2.2 The structure of a component skeleton 6

3 Defining the different aspects of a component 7

3.1 The parameters 7

3.2 The visual representation 8

3.3 The attachment areas 9

3.4 A factory for creating circles 10

3.5 The reference point 11

3.6 Testing your component 12

4 Manipulating the component 12

4.1 Standard handles 13

4.2 Special-purpose handles 14

5 Some more enhancements 15

5.1 User-definable default values 15

5.2 Serialization of components 16

6 Writing reusable components 17

7 Conclusion 19

Abstract

This tutorial shows how to define diagram components, which can be used in the *DiaGen* framework. The tutorial assumes that you have read the *general introduction* (as PDF or HTML) to the system and are familiar with the *DiaGen* architecture and some basic terminology.

The *Tree Tutorial* (as PDF or HTML) complements the Component Tutorial by explaining how to write a digram type specification and how to generate an executable diagram editor.

1 Aspects of a diagram component

Diagram components in *DiaGen* are represented as Java classes that must implement a number of interfaces; user-defined components should always be derived from the abstract convenience class `diagen.editor.lib.StandardComponent`, which combines all the necessary interfaces, or from a concrete component class that itself inherits from `StandardComponent`.

The following aspects must be defined in a concrete component class:

1. the parameters that characterize the component geometry
2. the visual representation that is displayed in the editor
3. the information that is needed to build the SRHG: the hypergraph edge label and the attachment areas
4. means of manipulating the component: handles and possibly component-specific editing actions (property dialogs etc.)

Most of these aspects are specified using polymorphic inheritance: The concrete component class has to implement or override methods of the base classes and interfaces. For the scanner aspect (item no. 3) this is done automatically by the generator module, to ensure that the information (e.g. the edge labels) in the component classes always matches the definition in the specification file; the other aspects must be coded "by hand".

Not all of the aspects need to be implemented in a single Java class; it is possible to build inheritance hierarchies of components where those aspects are defined in several steps, shared among similar components or redefined through overriding. In particular, it may be useful to separate the scanner information from the specification of parameters and manipulation properties: The latter can often be used for different diagram types (visual structures like circles, lines and boxes are useful for a lot of diagram types), while the former aspects (edge labels and attachment areas) usually depend heavily on the type of diagram in question.

As a comprehensive example, this tutorial will show you how to specify a simple circle component. You will need Java version 1.2 or higher and, of course, the *DiaGen* framework classes that are contained in the "diagen.jar" file. Make sure the jar-archive is contained in your Java class path. You should also create a scratch working directory and execute all commands from there. The ftp code directory for the tutorial contains all the source code file that you will be asked to create in the tutorial, so you can always copy them from there instead of typing them in yourself.

2 Component skeletons

When you write component classes for a specific diagram language, you do not start from scratch. Instead, you use the *DiaGen* generator module to create a skeleton source file. Then you edit that source file and insert code that describes the different aspects of the component.

Auto-generated skeletons should be used only for concrete components that are part of a single specific diagram type. If you intend to write a "library component" that is going to be used in more than a single diagram language, you should follow a different approach, which will be described later (see section 6)

2.1 Generating a component skeleton

When you define a new diagram language with *DiaGen* the first thing that you should think about are the components, the relations between them and the attachment areas that link components and relations. (If this is all greek to you, you should read the *general introduction* (as PDF or HTML) first.)

As soon as you have decided on those aspects, you can write a "minimal" diagram type specification with empty reducer and grammar specifications. For our example, we simply want to define a component with a "circle" label for the component edge and a single attachment area. It will be represented by a java class called "Circle" and the attachment area label is going to be "CircleArea".

[File from code directory: circle.spec]

```
component circle[1] { Circle[CircleArea] };

nonterminal Dummy[0];

reducer { }
grammar { start Dummy; }
```

Now process this specification file with the *DiaGen* generator to create a skeleton source file for the circle component:

```
java diagen.generator.Generator -f -s -p circle.spec
```

The -f flag tells the generator to overwrite existing source files, the -s flag causes the class skeletons to be created and the -p flag is necessary to turn off some optimizations, which would otherwise optimize the empty grammar away and cause run-time errors. You should get the following output:

```
WARNING: Start symbol Dummy is not reducible.
WARNING: The following symbols cannot be reduced to terminals:
  [_the_start, Dummy]
Writing Circle.java
```

You can ignore the two warnings, as we are currently only concerned with creating the component skeleton. You can find more information about writing complete and correct specification files in the *Tree Tutorial* (as PDF or HTML). If you have a look at you working directory now, you should find several new Java source files, among them a file “Circle.java” that contains the component skeleton.

2.2 The structure of a component skeleton

Now have a closer look at the “Circle.java” source file: You will find that it consists mainly of comments, which are intended to give you some help for fleshing out the skeletons. These comments contain notes for helping you with the coding (lines that start with a star) as well as actual code fragments that might be useful for defining the component, as in the following example:

[File from code directory: Circle1.java]

```
  /*
   * define the parameters and points here *
   public Parameter A, B, ...;
   public ParametricPoint2D P, ...;
   *
   * if you have more than one point, include this definition *
   protected Point2D[] points = new Point2D[N_PARAMS];
   *
   public static final int N_PARAMS = *edit here*;
   public int getNParams() { return N_PARAMS; }
  */
```

For the rest of this tutorial, we will ignore those comments, but when you follow the development of the circle component, you will find that most of the coding steps in this tutorial correspond to flehsing out one of those commented-out code fragments.

apart from the comments, the skeleton file contains a section that is framed by comments stating “start of auto-generated code” and “end of auto-generated code”:

[File from code directory: Circle1.java]

```
/* start of auto-generated code -- do not edit */

  public static final String NAME = "circle";
  public String getName() { return NAME; }

  public static final int N_ATTACHS = 1;
  public static final String[] ATTACH_NAMES = {
    "CircleArea"
  };
  public int getNAttachs() { return N_ATTACHS; }

/* end of auto-generated code */
```

Of course, the complete source file is auto-generated at this point, but the generator will write out the parts *above and below those commments* only when the skeleton is created for the first time. After that, you are required to edit those code sections and the generator will not touch them again.

The lines *inside the marker comments* on the other hand, are rewritten every time the generator is run again on the specification file ans the -s option is given. This ensures that this part of the component definition (which contains the scanner-related aspects) always corresponds to the definitions that are given in the specification file. Any changes that you make in this section will be lost when the generator is run again with the -s option.

3 Defining the different aspects of a component

Now we will start to convert the skeleton source into a working component. You can follow the steps by inserting the presented code fragments into the Circle.java source (and possibly deleting the comments if you like). Alternatively, the Circle2.java file from the code directory contains the complete source after executing those steps, so you do not have to edit the source by hand.

3.1 The parameters

The first step is to define the parameters that are needed to describe the shape of the component. Although it is possible to model dependencies between those parameters, this can easily lead to subtle problems with the editing behavior of the component. Therefore we recommend that you try to avoid redundant or dependent parameters. It is natural to describe a circle by its x and y center coordinates and its radius. When a pair of parameters is used to describe a point on the drawing plane, it is usually useful to define a `ParametricPoint2D` variable for them so the can be accessed as a `Point2D` unit.

The parameters are defined as member variables and initialized in the constructor. References to all parameter variables must also be stored in the `params` array; parameter references are not allowed to change. The number of parameters must be made accessible by implementing the `getNParams` method.

[File from code directory: Circle2.java]

```
public Parameter xc, yc, radius;
public ParametricPoint2D center;

public static final int N_PARAMS = 3;
public int getNParams() { return N_PARAMS; }

protected Circle(double x, double y, double r) {
  params[0] = this.xc = new Parameter("x-center", x, this);
  params[1] = this.yc = new Parameter("y-center", y, this);
  params[2] = this.radius = new Parameter("radius", r, this);
  center = new ParametricPoint2D(xc, yc);
}
```

If you want to access the parameters from the reducer and parser productions, you also need to define a public attribute accessor `get`*Attribute* for every parameter. This makes it possible e.g. to use the parameter values in dynamic layout constraints.

3.2 The visual representation

The visual representation is constructed from the parameter values in the `computeVRep` method. This method must return an object that implements the `VisualRepresentation` interface; i.e. it can be drawn on the screen, defines a bounding box and a hit testing method. The easiest way to construct a visual representation is to create a Shape object with the Java2D toolkit and convert it to the required interface with the `SimpleVRep` adapter.

[File from code directory: Circle2.java]

```
  protected VisualRepresentation computeVRep() {
    double xc = this.xc.getVal();
    double yc = this.yc.getVal();
    double radius = this.radius.getVal();
    Ellipse2D shape = new Ellipse2D.Double();
    shape.setFrame(xc-radius, yc-radius, 2*radius, 2*radius);
    return new SimpleVRep(shape);
  }
```

The first step is to read out the required parameter values and store them into local variables. The we create an `Ellipse2D` object, which implements the `Shape` interface and set the shape f the ellipse from the parameter values. Finally the shape is wrapped into a`SimpleVRep` and returned.

3.3 The attachment areas

Almost all of the code that defines the scanner information can be generated automatically from the specification file. The only thing that must be filled in by hand is the creation of the actual object instances for the attachment areas, because the generator cannot know which type of attachment area is required (i.e. how the bounding box is calculated). For the circle component, we can use the predefined `ShapeArea` class which represents an attachment area whose bonding box is equal to that of the corresponding component (resp. its visual representation). The creation of the attachment areas is handled by the `initAttachs` method.

[File from code directory: Circle2.java]

```
  protected void initAttachs() {
    attachs[0] = new ShapeArea(ATTACH_NAMES[0], this);
  }
```

We have found that the library classes `ShapeArea` and `PointArea` (which calculates the bounding box as a small square around a parametric point) are sufficient for defining most diagram languages; however, if you need something more specific, it is easy to define your own attachment area classes by subclassing `diagen.editor.graph.AttachmentArea` and redefining the `computeBounds` method.

3.4 A factory for creating circles

As the *DiaGen* editor classes need a way to create new components of a specific type, a diagram type must have a reference to a factory for every concrete component class that is allowed in the diagram. There should be exactly one factory per component type, so we follow the pattern of defining this factory as an anonymous local class inside the component class and making a single instance it available through a static member variable, which is named `FACTORY` by convention.

[File from code directory: Circle2.java]

```
public static final String[] PROMPTS = { "Circle center" };

public static final ComponentFactory FACTORY =
  new ComponentFactory(PROMPTS) {
    public DiagramComponent create(Model model, Point2D[] pos,
                                   Defaults defaults) {
      return new Circle(pos[0].getX(), pos[0].getY(), 20.0);
    }
};
```

A factory class must be a subclass of ComponentFactory and it must define a method `create` that returns a new component of the required type. Before the component is created, the user is asked to specify a number of points on the drawing plane, which are passed to the `create` method in order to initialize the shape of the component. The argument to the factory constructor (the `PROMPTS` constant) specifies how many points are needed by giving an array of prompt strings that are displayed while the user should select those points. The points are then passed to the`create` method in the `pos` argument. The `defaults` argument to the `create` method contains a map of user-definable default values (standard font, line width etc.) that can be used for intitializing the new component.

For the circle factory, we let the user specify the center position and set the radius the an initial value of 20.0. For other component types if may be more convenitent to let the user specify multiple points, e.g. the two endpoints for a line or two corners for a rectangle.

The user-interface properties of a component factory can be determined by calling its `setValue` method. For example, we should give a name to the factory, which will be displayed on UI objects that allow to create a new component:

[File from code directory: Circle2.java]

```
static {
  FACTORY.setValue(ComponentFactory.CREATE_NAME, "Circle");
}
```

In the same manner you can define an icon for the component and a property editor name and icon. Of course, if you define a property editor name, then you must also specify what happens if the user wants to access it; you do this by overriding the `createPropertyEditor` method.

3.5 The reference point

A component must provide support for translation by implementing methods to read and set an abstract "reference point". The position of the reference point relative to the component's visual representation can be chosen freely by the programmer. For the circle class, it is natural to take the center as the reference point:

[File from code directory: Circle2.java]

```
    public Point2D getRefPoint() { return center; }

    public void setRefPoint(Point2D refPoint) {
        center.setLocation(refPoint);
    }
```

Typically, all absolute coordinates of a component are given by its point parameters and it can be moved by just translating them while retaining their relative distances. The move-Points method of the StandardComponent class provides this functionality. The following code sample shows how this is done:

A point array must be declared that references all point parameters

```
    protected Point2D[] points = new Point2D[...];
```

It can be initialized in the constructor

```
    points[0] = pointParam1 ...
```

The implementation of the reference point is straightforward

```
    public Point2D getRefPoint() { return points[0]; }

    public void setRefPoint(Point2D refPoint) {
        movePoints(points, refPoint);
    }
```

To be able to compile the component class, you must also implement the abstract method`createHandles`. For the moment, you can just leave the method empty; we will come back to it later.

[File from code directory: Circle2.java]

```
public void createHandles(EditorPane panel) {}
```

3.6 Testing your component

By now the circle component is complete enough so that we can try to use it in an actual diagram editor. First, you have to compile the component class, which should hopefully produce no more errors at this point:

`javac Circle.java` (or `javac Circle2.java`, if you copied the file from the source directory)

You must also compile the three auto-generated source files that define the diagram type, the reducer and the parser:

```
javac DiagramType.java Transformer.java Grammar.java
```

DiaGen comes with a generic default editor that takes the name of a diagram type class as a command-line argument. You can use this editor to try out your component (make sure that the editor can find the DiagramType.class file):

```
java diagen.editor.ui.Editor DiagramType
```

You should see an editor window that shows, among other controls, a button that bears the name you just defined for the circle factory. When you press the button, you will see the prompt string in the window's status bar and whenever you click now on the drawing plane, a new circle will be created at that position. To leave the circle-creation mode click on the "Select mode" button.

When you are in "select" mode and click inside a component, you will notice that its color changes to red to indicate that it is now selected. Using the "edit" menu or the toolbar buttons, you can do copy, cut & paste with the selected component and you can also delete it again.

4 Manipulating the component

You can now create, display and delete components but you can not yet modify them in any way once they are placed on the drawing surface. To modify a component *DiaGen* provides the concept of "handles" (see the *general introduction*, as PDF or HTML). A handle is a draggable user-interface object whose position,

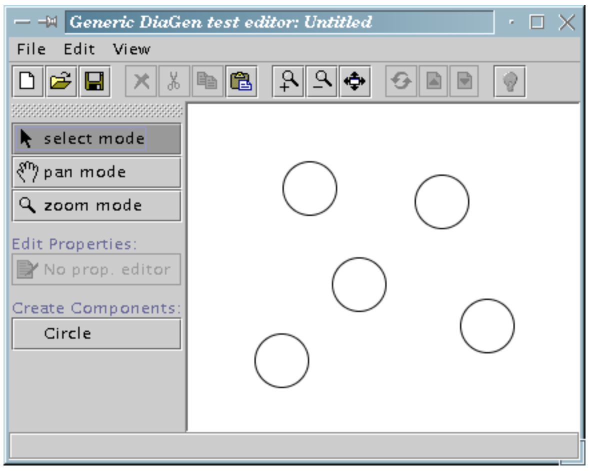

Figure 1: Screenshot of the editor

described by its reference point, is linked to the parameters of a component: If the handle is moved, the parameters are adjusted; if the parameters change usually because another handle was moved the handle s position is updated.

4.1 Standard handles

Every component type defines specific handles that are instantiated in the `createHandles` method. *DiaGen* provides some pre-defined handle classes. Examples:

- The `MoveHandle` class is displayed as a frame around an object and can be used to translate it to a different position. As every diagram component must support the `Movable` interface, you can always create a move handle for it.
- The `PointHandle` class represents a handle that is displayed a a small black square. Its position is linked to a `Point2D` object (typically a `ParametricPoint2D`).

For the circle component, we can use a point handle to modify the center coordinates, and we need a special handle class to manipulate the radius (the class will be described in the next section). We also give the component a move handle, although that is redundant because its function is identical to moving the center.

[File from code directory: Circle3.java]

```
public void createHandles(EditorPane panel) {
  panel.addHandle(new PointHandle(this, center, panel));
  panel.addHandle(new RadiusHandle(panel));
  panel.addHandle(new MoveHandle(this, panel));
}
```

The order in which the handles are created is important if they overlap on the display, because it determines which handle is selected by an ambiguous mouse click (the one that was created first): In our case, the radius handle overlaps the move handle frame and it would be quite hard to select it if the move handle was created first.

4.2 Special-purpose handles

In many cases, the library handle classes will not be flexible enough to support the desired editing operations. It is easy to define local handle classes that define special relations between the handle's position and the parameter values of the component. An example for this is the handle that changes the radius of the circle.

[File from code directory: Circle3.java]

```
private class RadiusHandle extends BoxHandle {
  RadiusHandle(EditorPane parent) {
    super(Circle.this, parent);
  }
  public Point2D computeRefPoint() {
    return new Point2D.Double(xc.getVal()+radius.getVal(), yc.getVal());
  }
  protected void updateParams() {
    radius.setVal(this.getRefPoint().distance(center));
  }
```

The graphic representation of the radius handle as a small black square is provided by the superclass `BoxHandle`. As you can see, all that remains to do is to specify how the position of the handle can be computed from the component parameters (by implementing the `computeRefPoint` method) and how the parameters change if the handle is dragged to a new reference point (by implementing the `updateParams` method)

You can now compile the modified component class and try the generic test editor again. When you now select a component, you should see its handles on the screen and by dragging them you should be able to manipulate the circles.

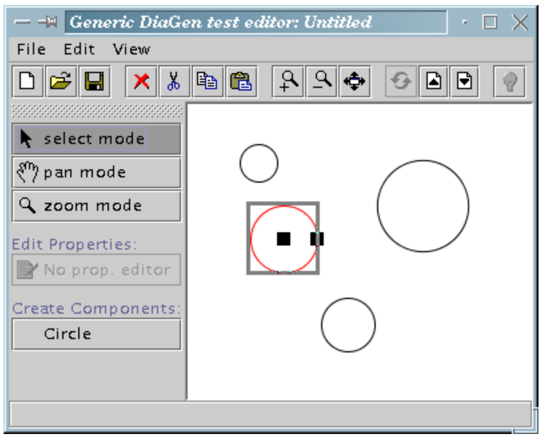

Figure 2: Screenshot with active handles

5 Some more enhancements

The circle component is now complete and usable. The following section will now present some more features of *DiaGen* components by modifying our example component a little.

5.1 User-definable default values

When we defined the factory for the circle components, we chose to “hard-code” the initial radius value to 20. In some cases it might be useful to let the user modify this initial default setting. Similarly, many drawing programs provide default settings for properties like line widths, text fonts and sizes, arrow shapes and the like. *DiaGen* editors provide a standard way to access such default values through a `Defaults` object that is passed to the `create` method of a factory. This object stores such default values and you can index them with arbitrary strings.

A modified circle factory that uses a default value for the initial radius would look like this:

[File from code directory: Circle4.java]

```
public static String RADIUS = "defaults.circle.radius";

public static final ComponentFactory FACTORY =
  new ComponentFactory(PROMPTS) {
    public DiagramComponent create(Model model, Point2D[] pos,
                                   Defaults defaults) {
      return new Circle(pos[0].getX(), pos[0].getY(),
                        defaults.get(RADIUS,20.0));
    }
};
```

As it is possible, that no default radius value is defined, you should give an alternative value to the `get` method, which will be used in that case.

The generic editor does not currently provide a way to set those default values. To be able to set the default radius, you would have to extend the `Editor` class or provide your own main editor.

5.2 Serialization of components

The save/load functionality of *DiaGen* editors uses the standard Java serialization mechanism. This means that any member variables that you define for a component will be written out and read from a file. In most cases, this is what you want: Parameters should always be serializable; fields that cannot be recreated automatically, for example text labels that can be edited by the user, should be serializable as well. But member variables should instead be marked as `transient`, if it is possible to regenerate them when the component is loaded. This refers for example to fields that hold cached values from expensive computations. Transient fields can be initialized by overriding and extending the component's `register` method, because this method must always be called before the component can be used in any way.

To give a short example, we introduce a transient member into the circle component: If you look at the code for the `computeVRep` method (see section 3.2, you will see that it always creates a temporary `Ellipse2D` object. We could avoid the overhead for allocating a new object (which is not quite compelling in this case) by creating one single ellipse for the entire lifetime of the component and keeping it as a member variable. Obviously, this member does not need to be serialized.

[File from code directory: Circle4.java]

```
protected transient Ellipse2D shape;
```

Now we need to overload the `register` method in order to intialize the member variable.

[File from code directory: Circle4.java]

```
public void register(Model model) {
  shape = new Ellipse2D.Double();
  super.register(model);
}
```

You must *never* use initializers for transient member variables; your objects will work just fine when they are created by a constructor call, but when the object is loaded from a stream, the members will be set to Java's default values (usually `null`) which usually results in a crash.

Now you can delete the line that creates a new ellipse from the `computeVRep` method:

[File from code directory: Circle4.java]

```
protected VisualRepresentation computeVRep() {
  double xc = this.xc.getVal();
  double yc = this.yc.getVal();
  double radius = this.radius.getVal();
  shape.setFrame(xc-radius, yc-radius, 2*radius, 2*radius);
  return new SimpleVRep(shape);
}
```

6 Writing reusable components

You will sometimes want to reuse the same type of component in different diagram languages. Circles make a good example of "general purpose" components that can be used in a lot of different contexts. The *DiaGen* generator demands that concrete component classes are always defined in the same package (and directory) as the diagram type so it can find and modify the source files easily. Therefore if you have defined a component for one diagram type that you want to use in another, we recommend that you split the copmonent into separate classes:

- A *concrete compoponent class* for every specific diagram type that uses the component. The concrete component should only contain the attachment area definitions and a factory. All the concrete components inherit most of their functionality from

- a single abstract *library component* that contains the code for all other aspects of the component, i.e. the parameters, visual representation, handles, special property editors etc. This class should be placed in a different package where it is globally accessible.

To give an example we have applied this pattern to the circle component that was defined in this tutorial.

The library component could look like this:

[File from code directory: LibCircle.java]

```
package foo;

import ...

public abstract class Circle extends diagen.editor.lib.StandardComponent {

  public Parameter ...

  protected Circle(double x, double y, double r) { ... }

  protected VisualRepresentation computeVRep() { ... }

  public Point2D getRefPoint() { ... }
  public void setRefPoint(Point2D refPoint) { ... }

  public void createHandles(EditorPane panel) { ... }

  private class RadiusHandle extends BoxHandle { ... }
}
```

A concrete component built from this library component would then contain the following code parts:

[File from code directory: UseCircle.java]

```
class Circle extends foo.Circle {

  protected Circle(double x, double y, double r) {
    super(x,y,r);
  }

/* start of auto-generated code -- do not edit */
...
/* end of auto-generated code */

  protected void initAttachs() { ... }

  public static final ComponentFactory FACTORY =
    new ComponentFactory(PROMPTS) {
    ...
  };

  static {
    FACTORY.setValue(ComponentFactory.CREATE_NAME, "Circle");
  }

}
```

7 Conclusion

Hopefully this tutorial has given you some help for coding the components that you need for your use of *DiaGen*We have covered most of the coding patterns and library classes that can help you with this. For further information you can have a look at the sample editors that are included in the *DiaGen* distribution. (You should keep in mind, though, that those examples are not always written in the most elegant way that one could imagine.) The *JavaDoc documentation* of the *DiaGen* framework is also accessible on the Web; you will find some more information about the library classes there.