Extending Kandid

This is a small tutorial about extending Kandid with a new types of image producing calculations.

0.1 Kandid

Kandid is a system to evolve graphics. Graphics, in Kandid, is not drawn by hand. Instead new forms can be found using genetic algorithms. To achieve this aim Kandid simulates evolution using sexual reproduction and populations. But there is no fitness function in side the program. Only the user decide which images are interesting.

0.2 Kandid on the Net

Kandid homepage

http://kandid.sourceforge.net/

0.3 Extending Kandid

Kandid has two different types of image calculations. In the simpler case the image rendering code is written by an human and only the parameters, feeding this algorithm, comes from a genetic system. This is discussed in this paper.

In other types of calculation the rendering code is evolved with a genetic system too. This will be discussed in an extra paper.

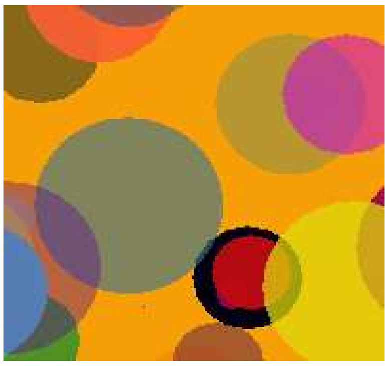
0.4 Steps for extending Kandid

- Define the chromosomes data model in the soup.xsd schema file.
- Run the code generator to produce genetic operators, persistence and other helper routines for the new chromosome type.
- Subclass one oft the calculation base classes and implement the image renderer.
- Add the new calculation type to the catalog.xml file.
- Recompile and restart Kandid

1. Adding a simple new Population

1.1 The shape example

This example population produces images with coloured circles. The circles will overlap, but they are transparent.



1.1.1 Data model

An image population consists of two parts. Part one is the data model with all the parameters that should be evolved. The second part is the calculation of the images based on the parameters. While the image calculation is hand coded for every type of calculation the parameter evolver should be implemented in a more general way.

Modelling the parameters

The core of the parameter evolving system is a data model written in XML schema. It is located in file kandid/soup/schema/soup.xsd. This is the point where we will start with the example. The only shapes we have at the moment are circles. All of this shapes have a colour, a transparency, a dimension and a position. In one image is more than one shape that should be painted on a solid filled background.

For modelling the shaped we will use inheritance. Shape is an gene in our model with a colour attribute and a position in 2D coordinate space.

```
 <xsd:complexType name="abstractTutorialShapeGene">
  <xsd:complexContent>
    <xsd:extension base="geneType">
      <xsd:sequence>
        <xsd:element name="color" type="colorGene"/>
        <xsd:element name="transparency" type="normalizedGene"/>
        <xsd:element name="x" type="symmetricGene"/>
        <xsd:element name="y" type="symmetricGene"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

The <xsd:extension base="geneType"> is mandatory for all genes. This tag told Kandid to handle shape as a gene. The attribute are modelled with the <xsd:element/> tags. In this case types of all attributes are based on predefined types. This predefined types have a given value range, mutating and random seed behaviour. Type colorGene has red, green, blue components, normalizedGene stands for values between 0.0 and 1.0 and symmerticGene stands for the range -1.0 to +1.0.

Now we can define our visible shape type, the circle.

```
 <xsd:complexType name="tutorialCircleGene">
  <xsd:complexContent>
    <xsd:extension base="abstractTutorialShapeGene">
      <xsd:sequence>
        <xsd:element name="radius" type="normalizedGene"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

TutorialCircleGene inherit from AbstractTutorialShapeGene adding its own attribute, the radius.

What we need is an list of of circles and not al list of abstract shapes. We can do this with minOccurs and maxOccurs attributes of XML Schema.

Every type of calculation needs one chromosome defining the parameter set. We are nearly ready expanding the core of the data model. We model the list of shapes that we will be later painted on a non transparent background.

```
<xsd:complexType name="tutorialShapeChromosome">
  <xsd:complexContent>
    <xsd:extension base="chromosomeType">
      <xsd:sequence>
        <xsd:element name="allShapes" type="tutorialCircleGene" minOccurs="2" maxOccurs="100"/>
        <xsd:element name="color" type="colorGene"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Setting both minOccurs and maxOccurs told Kandid that a list of genes is required. The <xsd:choice/> tag means that circles and rectangles are an alternative.

At the end of the data modelling some technical code must be added to the soup.xsd schema.

You must add an definition for the new image type.

```
<xsd:complexType name="shapeImage">
  <xsd:complexContent>
    <xsd:extension base="imageType">
      <xsd:sequence>
        <xsd:element name="chromosome" type="shapeChromosome"/>
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:string" use="required" fixed="0.3.2"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

And you had to expand two tags at the beginning of the file to insert your image type to the other population types. Data type Entity is for storing single chromosomes. This is used when exporting images and for images in the pool.

```
<xsd:complexType name="entityType">
  <xsd:choice>
         ...
         ...
         ...
    <xsd:element name="shapeImage" type="shapeImage"/>
  </xsd:choice>
         ...
</xsd:complexType>
```

Data type Population is for storing whole populations.

```
<xsd:element name="population" type="populationType"/>

<xsd:complexType name="populationType">
  <xsd:choice>
         ...
    <xsd:element name="shape" type="shapeImage" maxOccurs="unbounded"/>
  </xsd:choice>
         ...
</xsd:complexType>
```

Auto generating code

But why all this abstract meta data definitions? What is the advantage? Dependent on your data model you need a lot of stupid code. The parameters should be stored and loaded from file, should be used in an drag and drop operation, the user wants to see or edit the values, chromosomes must be initialised with random values, or mutated or merged together... And all of this code can be generated automatically from the schema. You must only start the shell script ./schema to produce this code. If you are working with an IDE you had to refresh the project, because an IDE like Eclipse had to reimport the changes.

There are two slightly different pieces of generated code. Code for storing, loading and schema verification is generated with the Java Architecture for XML Binding (JAXB). Code specific to genetics algorithm is generated with XSLT. There are some XSL files to produce Java source code from the soup.xsd schema. The decision made in this XSL file are often based on naming conventions. The name of genes or chromosomes must always end with the phrase “Gene” or “Chromosome” . For example type="colorGene".

Here is an example of an piece of auto generated code for initializing a chromosome with a list of shapes. It generates a list with at least 2 and not more than 100 rectangles or circles.

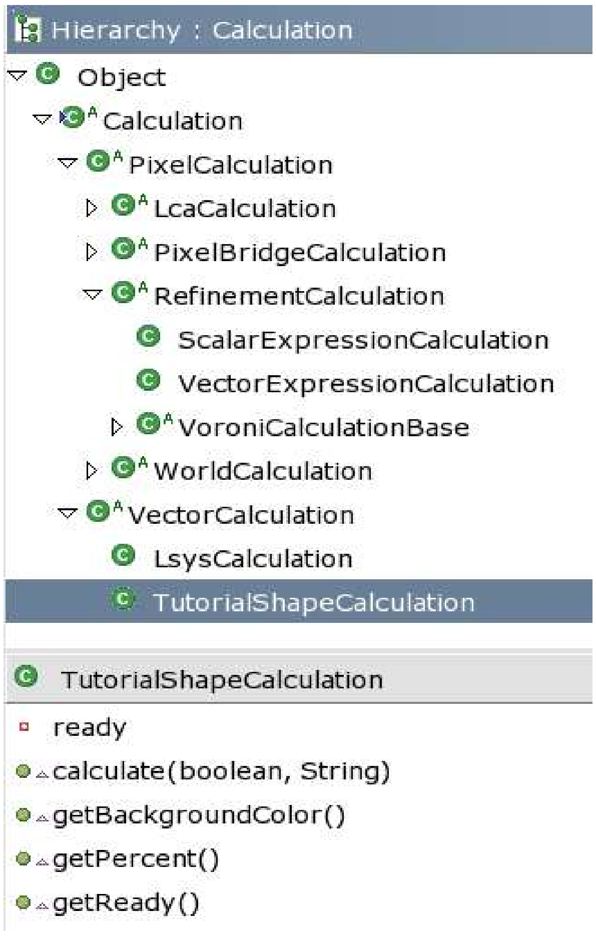
```
public void randomize(TutorialShapeChromosomeImpl aChromosome)
                                                    throws JAXBException {
  {
    java.util.List list = aChromosome.getAllShapes();
    list.clear();
    int newLength = kandid.util.CentralRandomizer.getInt(2, 100);
    while (newLength > 0) {
      TutorialCircleGene aAllShapes = objectFactory.createTutorialCircleGene();
      randomize(aAllShapes, aChromosome, 1);
      list.add(aAllShapes);
      --newLength;
    }
  }
  ColorGene aColor = objectFactory.createColorGene();
  randomize(aColor, aChromosome, 1);
  aChromosome.setColor(aColor);
}
```

This generated code is triggered by the following tags:

```
<xsd:sequence>
  <xsd:element name="allShapes" type="tutorialCircleGene" minOccurs="2" maxOccurs="100"/>
  <xsd:element name="color" type="colorGene"/>
</xsd:sequence>
```

1.1.2 Image calculation

Different Calculation classes can be used as a base for painting shape image.


Hierarchy : Calculation

```
Object
  Calculation
    PixelCalculation
      LcaCalculation
      PixelBridgeCalculation
      RefinementCalculation
        ScalarExpressionCalculation
        VectorExpressionCalculation
        VoroniCalculationBase
      WorldCalculation
    VectorCalculation
      LsysCalculation
      TutorialShapeCalculation
```

```
TutorialShapeCalculation
  ready
  calculate(boolean, String)
  getBackgroundColor()
  getPercent()
  getReady()
```

PixelCalculation is the base class for calculation for raster graphics. The calculation has to evaluate the colour of every pixel. Non of the real calculation is directly derived from this base class.

RefinementCalculation is used for calculations able for producing low resolution previews. This will give a faster feedback to the user. The Voroni calculation is such a kind of calculation.

PixelBridgeCalculation is used when the calculation of the image is done outside of Kandid. This is at the moment in an experimental state.

VectorCalculation is for vector graphics. The image must be constructed with java.awt.Graphics2D methods. For screen and bitmap export the draw and fill methods produces bitmap data. This is done by the AWT graphics engine. With the help of the Batik packages from apache.org the java.awt.Graphics2D methods can be mapped to SVG.

Coding

The shape example sounds like vector graphics. We will create the new class `TutorialShapeCalculation` derived from `VectorCalculation` in the new `kandid.calculation.tutorial.shape` package . This can be done with Eclipse very easy. Eclipse overwrites the abstract methods from VectorCalculation with stubs.

```
package kandid.calculation.tutorial.shape;

import kandid.calculation.VectorCalculation;
```

```
/**
 * @author thomas jourdan
 *
 */
public class TutorialShapeCalculation extends VectorCalculation {

  /* (non-Javadoc)
   * @see kandid.calculation.Calculation#calculate(boolean, java.lang.String)
   */
  public void calculate(boolean preview, String exportFilename) {
    // TODO Auto-generated method stub

  }

  /* (non-Javadoc)
   * @see kandid.calculation.Calculation#getPercent()
   */
  public int getPercent() {
    // TODO Auto-generated method stub
    return 0;
  }

  /* (non-Javadoc)
   * @see kandid.calculation.Calculation#getReady()
   */
  public boolean getReady() {
    // TODO Auto-generated method stub
    return false;
  }

  /* (non-Javadoc)
   * @see kandid.calculation.Calculation#getBackgroundColor()
   */
  public int getBackgroundColor() {
    // TODO Auto-generated method stub
    return 0;
  }

}
```

Two of this method stubs can be fixed in a simple way. We don't need the percentage display and we set the default background to black.

```
  public int getPercent() {
    // -1 disables the percentage display, that's Ok for the first step.
    return -1;
  }

  /* (non-Javadoc)
   * @see kandid.calculation.Calculation#getBackgroundColor()
   */
  public int getBackgroundColor() {
    // 0 means black background.
    return 0;
  }
```

For an first step we will simplify the calculation. We will only paint the background colour provided from the parameter set. After filling the background the calculation is ready. This is signalled with a getReady() returning true.

```
  public void calculate(boolean preview, String exportFilename) {
    // get chromosome
    TutorialShapeChromosome shapeChromosome =
                              (TutorialShapeChromosome)chromosome;
    // initialize transformation without border
    initTransformation(0.0);
    // get the background color and fill the whole canvas
    Color background = new Color(shapeChromosome.getColor().getRed(),
                             shapeChromosome.getColor().getGreen(),
                             shapeChromosome.getColor().getBlue());
    g2d.setColor(background);
    g2d.fillRect(0, 0, canvasSize.width, canvasSize.height);

    // image is complete
    ready = true;
  }

  public boolean getReady() {
    return ready;
  }
```

The chromosome is a member variable located in a base class. We had to down cast it to our specific chromosome type. Then the chromosomes parameters can be accessed and must be mapped to Graphics2D method calls.

1.1.3 Catalogue

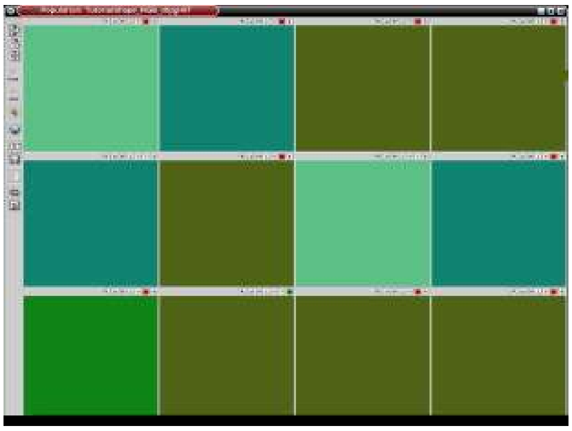
Adding the new type

Kandid reads an catalogue located in kandid/catalog/catalog.xml. Before the new calculation can be used it must be listed in this catalog.

```
<calculation name="TutorialShape">
  <model>
    <calculationClass>kandid.calculation.tutorial.shape.TutorialShapeCalculation</calculationClass>
    <coloratorClass>kandid.colorator.RGBColorator</coloratorClass>
  </model>
  <icon url="/img/typeShapeTutorial.png" gridX="2" gridY="3"/>
  <tooltip>Tutorial example. How to add a new type of population to Kandid</tooltip>
</calculation>
```

There are two rules: The name in the <calculation name="TutorialShape"> tag comes from the name of the image defined earlier in the soup.xsd shema. The postfix “Image” must be removed and the first letter must be capitalized. <xsd:complexType name="tutorialShapeImage"> becomes <calculation name="TutorialShape”> Second you had to provide an image icon with an 64x64 dimension and place it with the gridX and grodY attribute. This image should reside in the img folder.

Now its possible to start Kandid the first time with the new calculation.



1.1.4 Completing the shape calculation

Filling the shapes

Kandid is now able to display the generated images in the population frame. With this visual feedback it is easier to complete the shape calculation. As we modelled in the schema the shapes come from a list having colour, transparency, dimension and location attributes.

```
public void calculate(boolean preview, String exportFilename) {
  // get chromosome
  TutorialShapeChromosome shapeChromosome =
                               (TutorialShapeChromosome) chromosome;
  // initialize transformation without border
  initTransformation(0.0);
  // get the background color and fill the whole canvas
  Color background =
    new Color(shapeChromosome.getColor().getRed(),
              shapeChromosome.getColor().getGreen(),
              shapeChromosome.getColor().getBlue());
  g2d.setColor(background);
  g2d.fillRect(0, 0, canvasSize.width, canvasSize.height);

  List allShapes = shapeChromosome.getAllShapes();
  for (Iterator shapeIter = allShapes.iterator(); shapeIter.hasNext();) {
    Object element = shapeIter.next();
    if (element instanceof TutorialCircleGene) {
      TutorialCircleGene shape = (TutorialCircleGene) element;
      // get the fill color
      Color fillcolor = new Color(shape.getColor().getRed(),
                                  shape.getColor().getGreen(),
                                  shape.getColor().getBlue());
      g2d.setColor(fillcolor);
      // get transparence and map to Graphic2D alpha composite class
      float alpha = (float) shape.getTransparency().getValue();
      AlphaComposite ac =
            AlphaComposite.getInstance(AlphaComposite.SRC_OVER,alpha);
      g2d.setComposite(ac);
      // get position and transform
      transform(shape.getX().getValue(), shape.getY().getValue());
      int x1 = xp;
      int y1 = yp;
      // transform the radius and fill it
      transform(0.5*shape.getRadius().getValue(),
                0.5*shape.getRadius().getValue());
      int r1 = xp;
      // fill circle
      g2d.fillOval(x1+r1, y1+r1, r1, r1);
    }
  }
```

2. Calculations for colouring pixels

Some types of calculation formulas produces a sequence of points in 2D space and a colour information for each of this points. Iterated function systems (IFS) are an good example.

2.1 Integrating IFS

2.1.1 IFS data model

An affine Iterated Function System consist of an list of transformation matrices. Each matrix has six values. XML schema in soup.xsd models this.

```
<xsd:complexType name="affineTransformationGene">
  <xsd:complexContent>
    <xsd:extension base="geneType">
      <xsd:sequence>
        <xsd:element name="ta" type="symmetricGene"/>
        <xsd:element name="tb" type="symmetricGene"/>
        <xsd:element name="tc" type="symmetricGene"/>
        <xsd:element name="td" type="symmetricGene"/>
        <xsd:element name="te" type="symmetricGene"/>
        <xsd:element name="tf" type="symmetricGene"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<xsd:complexType name="affineIfsChromosome">
  <xsd:complexContent>
    <xsd:extension base="chromosomeType">
      <xsd:sequence>
        <xsd:element name="seed" type="seedGene"/>
        <xsd:element name="affineTransformation" type="affineTransformationGene"
                     minOccurs="2" maxOccurs="8"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

2.1.2 IFS Calculation

Looking a little bit deeper to pixel based computer graphics we see that the painting canvas has a fixed amount of discrete pixels. For example 265 in each direction. But the IFS produces points in a 2D areas with unknown bounds. Its is necessary to make one test run for every new image to calculate the bounds. Then you can set up the transformations mapping from IFS coordinates to graphic device coordinates. Every iterated point produced by the IFS must be transformed. This things are done in class In this class WorldCalculation. There are two member variables xw an yw representing the world coordinates. The coordinates produced during one iteration must be assigned to this variables. Then the method setWPixel(color) colours the appropriate pixel in the graphics device. This is not what you expected in object oriented programming. The design of this classes is influenced from performance requirements.

```
double x2 = xw * mTa[tx] + yw * mTb[tx] + mTe[tx];
double y2 = xw * mTc[tx] + yw * mTd[tx] + mTf[tx];
xw = x2;
yw = y2;
setWPixel(pointColor);
```

For a real example please see class WorldCalculation, IterationCalculation and

AffineIfsCalculation.

2.1.3 Incremental image calculation

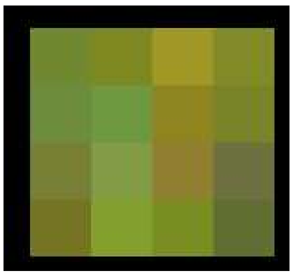
Even on fast computers most of the calculations implemented in Kandid needs a lot of time. The user may be more satisfied if it is possible to display preliminary images while the calculation is not finished. This is implemented in the following way. The Kandid framework calls the calculate() method of the specifics Calculation class. In side of this method some pixels are set to the calculated colours. But the calculation routine can be exited long before all iterations are ready and the Kandid framework paints the preliminary buffered image. As long getReady() returns false the calculation() Method will be called again from the framework to calculate the next round.

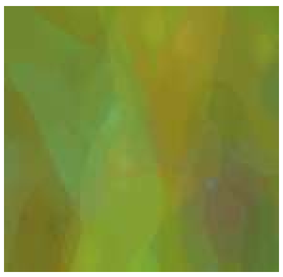
2.1.4 Successive refinements

Some types of calculations works like mathematical functions, where a result is produced dependent on the input parameters. There are no inner states and the same parameter values produce always the same result. Some of this formulas take a 2D coordinate and produce an colour value. This behaviour can be used for quick previews.

2.1.5 Voroini calculation

In a Vorni diagram each point in 2D space is coloured dependant of his distance to some control point. For the final image you had to calculate every pixel. But for an quick preview you can make greater steps reducing the calculation cost. But you had to fill rectangular areas instead of pixels. Class RefinementCalculation did this job in an intelligent way. It makes several runs decreasing the stepping distance from run to run until it comes to the final pixel by pixel rendering.





For a real example please see class RefinementCalculation and VoroniBaseCalculation.

2.2 Using external image generators

There are good image generators available to produce excellent images. It may be a solution to port there code to Java and the Kandid software architecture. But for example the Persistence of Vision ray tracer is a great thing developed over the last 10 years. A better approach was to use POV ray as it is.

Kandid can be used as an wrapper for image generation programs. These external programs must be able to read the parameters from a file or command line and produce the resulting image to an other file or to the stdout stream. It is not necessary that the user edits the parameter files by hand or starts the program manually. From the users view there is no distinction between build in and external calculations. But for programmers there are a lot of difference between internal calculation and external renderers. Normally these programs are written in C or C++ and compiled. Kandid has no access to there internal states and functions. This is the reason why Kandid produces in a first step the input parameter file, starts then external program invisible for the user, waits until the output is generated and display in a last step this output inside a normal

Kandid population frame.

A simple way collecting the output is giving the renderer an output file name and later loading this file. But with this strategy no preview can be shown to the user. An other way is to establish a data pipe between Kandid and the Renderer. This is done in the Kandid / POV ray bridge. Kandid told POV ray to produce his image to the stdout stream. During the external rendering Kandid reads the image back line by line in an parallel running thread.

For a real example please see class Bridge, FlameBridge and PovThingBridge. The Flame bridge uses the simple file based coupling. At the moment Persistence of Vision ray tracer and Scott Draves Flame IFS renderer is supported.

3.Software tools

3.1 Required tools

J2SE 1.4.x SDK or JDK 5.0 from SUN Microsystems

http://java.sun.com/

Java Architecture for XML Binding from SUN Microsystems

You can download the Java Web Services Developer Pack 1.3

Only the JAXB component is needed.

http://java.sun.com/webservices/webservicespack.html

Xalan-Java version 2.4.x from Apache.org

Xalan is the XSLT processor generating all the helper code from the soup.xsd schema.

http://xml.apache.org/xalan-j/

Batik 1.5.x from Apache.org

Batik is only used for exporting Images based on class VectorCalculation.

http://xml.apache.org/batik/

3.2 Recommended tools

Eclipse 3.x

The recommended IDE is Eclipse 3.x

http://www.eclipse.org/

If you change Java source code inside Eclipse and store it then this code is automatically compiled and Eclipse will try to update the code inside the running Kandid program. If this hot updating fails you got an error message and you had to restart Kandid using the "debug" button.

JUnit 3.8.x

If you have Eclipse installed it is not necessary to download Junit. It is already a part of Eclipse.

http://junit.org/

Ant 1.5.x from Apache.org

http://jakarta.apache.org/ant/

Schema expansion and packing the Kandid software is done with shell scripts executed on Linux with bash. If you are developing under Windows you must convert the shell scripts to batch files. A better way may be using ant from apache.org. At the moment only a few things are ported to ant. One advantage of ant is that its platform independent.

If you want to build a new kandid.jar code archive go to folder /kandid/src and type ant on your command line. Ant uses build.xml to compile a new version.