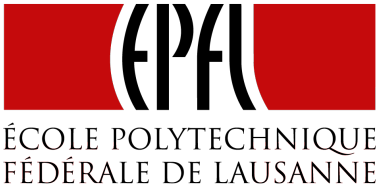

SHotDraw

Guillaume ULRICH

January 12, 2013

1 Introduction

Once upon a time, computer users were to be expert in order to use the old command line systems. But those days are gone. Nowadays, everybody wants its own computer and doesn't want to bother with command lines and terminals. This leads to a constantly growing need for interactive applications and graphical user interfaces. However, the pattern we use to handle with continuous input and output is still the observer-pattern, which is error prone and may be hard to implement. But things are about to change, for instance with Scala.React[1], a framework helping migrate from observer-based to more declarative logic.

We will use the 6^{th} version of the drawing tool JHotDraw[2] to experiment with the observer-pattern, the issues it may cause and some way to replace it. JHotDraw is a nice open-source drawing tool, that allows the user to draw figures such as rectangles, circles or polygons and link them using arrows. This is not a very powerful tool, but it is easy enough to be able to modify the code and complex enough to use a lot of event handling.

As the libraries we aim to use are in Scala, the first step is to translate the JHotDraw from Java to Scala. Of course, the best way to have a clean working Scala version would be to code everything from scratch, but this is unrealistic. As there are more than 30000 lines of code, we will use an automated translation to get a first version, but this is of course not sufficient to have a clean working Scala version; most of the code then has to be changed by hand.

With a first version of SHotDraw, we are able to start replacing the observers. As seen above, one of the way would be to use the Scala.React[1] framework. Another way to solve this problem is to use a constraint system solver, Cassowary[4], because it is the most efficient in the case of interactive application.

2 JHotDraw

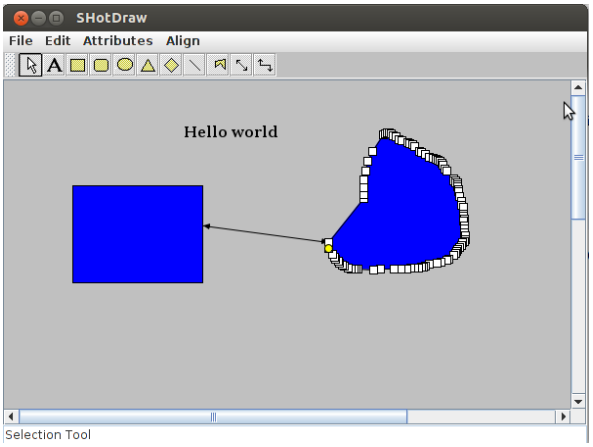

Figure 1: HotDraw preview

JHotDraw[2] is an open-source Java GUI framework for technical and structured Graphics. It offers multiple tools to create drawings. First there is a creation tool, offering the opportunity to create different figures, like rectangle, round rectangle, line, ellipse or more complex figures like the polygon figure, which has no defined shape, but the one drawn by the user or the text figure. It is also possible to create arrows connecting two figures together.

Furthermore, it is possible to change the figures attribute using the attribute menu. The attributes vary depending on a figure, you can change the border and fill color of plain figures, text font, size, name and color of text figures and arrow color and style. There is also an edit menu allowing to delete, duplicate, group or set the figure position in the drawing, back or front.

Finally, every action is undoable, except the figure creation, which is a pity, but will be changed in SHotDraw.

JHotDraw is an excellent choice, because it is open-source, not too complex but it has many interesting features. Thus, it is possible to understand how it works and translate it to Scala in a reasonable amount of time, but still powerful enough to use a lot of observers. Hence it is very good to experiment a way to use Scala.React or constraints instead.

The very first step of the project was to remove the useless code. Indeed, the JHotDraw source contains more than one application. In addition to the drawing application, there are a network application and a diagram application. As we want to concentrate on the drawing part, we don't need the other applications and we can remove the corresponding code.

There is one other important change that we had to make: changing the way collections were used. Indeed, the first versions of JHotDraw were released long ago and collections were not the way they are today. This is the reason why JHotDraw uses a collection factory. Nowadays, such a factory is deprecated and we can just replace it with actual java collections.

Finally, the last thing to do was to get to know how the code works. The application is composed of an editor containing a view and a set of commands. The view essentially manages all the figures; addition, deletion or selection. The commands consist of actions that can be executed on the view, like for instance the creation command, allowing one to create a new figure, which means add a new figure to the view, or the change attribute command. Most of the commands are encapsulated in an undo command that defines how an action should be undone or redone.

In order to get a better understanding of how things work, we added two new shapes: a triangle and a diamond. These figures are defined by a rectangle display box which is defined by its top left corner and bottom right corner. From the display box, we can compute everything; the handles are drawn along the display box, the shape of the figure is created from the origin, width and height of the display box and the default connector (AbstractConnector) connects along the display box.

A part of the TriangleFigure code.

```
public class TriangleFigure extends
    AttributeFigure {
    [...]
    private Rectangle fDisplayBox;
    [...]
    public TriangleFigure(Point origin,
        Point corner) {
    public void drawFrame(Graphics g) {
        //Similar to drawBackground with
            g2d.draw(path); in the end
    }
                basicDisplayBox(origin,corner);
    }

        public HandleEnumeration handles() {
        List<Handle> handles = new
            ArrayList<Handle>();
        BoxHandleKit.addHandles(this,
            handles);
        return new
            HandleEnumerator(handles);
    }

    public void basicDisplayBox(Point
        origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }
    [...]
    protected void basicMoveBy(int x, int
        y) {
        fDisplayBox.translate(x,y);
```

```
    }
    [...]
    public void drawBackground(Graphics g) {
        Rectangle r = displayBox();
        Graphics2D g2d = (Graphics2D)g;
        GeneralPath path = new
            GeneralPath();

        path.moveTo(r.x+r.width/2.0, r.y);
        path.lineTo(r.x, r.y+r.height);
        path.lineTo(r.x+r.width,
            r.y+r.height);
        path.closePath();

        g2d.fill(path);
    }

    public void drawFrame(Graphics g) {
        //Similar to drawBackground with
            g2d.draw(path); in the end
    }
    [...]
    public Connector connectorAt(int x, int
        y) {
        return new AbstractConnector(this)
            {};
    }
    [...]
}
```

On that piece of code, we can see a design mistake. Indeed, lots of figures are defined by a rectangle display box and in the end, the only thing that vary from one figure to another is the shape and the connector. So we could have one superclass managing the display box, the handles and the move operation and the subclass would only implement the drawBackground, drawFrame and connectorAt methods.

3 SHotDraw

From the cleaner and lighter version of JHotDraw, we will start the biggest part of the project: the translation. What we aim to have in the end is a Scala version of HotDraw, SHotDraw and from that version, see what we can do to remove observers and add new features. Hence the biggest part of the project is to translate JHotDraw from Java into Scala to obtain SHotDraw. As there are a lot of code to be translated, more than 30k lines of code, it is unrealistic to recode everything from the scratch. We will then translate the Java code line by line into Scala, starting with the classes that have no dependencies until we reach the class depending on all the other.

3.1 Using IntelliJ

This is a pretty long work and it would be easier to have an automated translation. But is there a way to translate automatically such a big Java program into a working Scala version ? No there is not. However, if you create a .scala file in IntelliJ[5] and copy/paste the Java code, the code is translated. Even though the translation is dirty and not compiling, it is still a translation from Java to Scala.

One big advantage of such translation is that we spare time on the variables. Indeed, in Java we write $intvariable = value$; when in Scala we write $varvariable : Int = value$, so translating variables it essentially exchanging the type and name positions. Another advantage is that we now have a Scala basis, very useful for the rest of the translation.

3.2 Problems with IntelliJ

There are not only advantages using IntelliJ, there are also a bunch of problems. First of all, the code doesn't compile. But once we have solved the compilation issues, still there are runtime errors and the code is full of dirty parts. For instance, the return used in every non void method in Java is not necessary in Scala.

A major problem appears with every for loop. The following Java for loop:

```
for(int i = 0; i < array.length; i++) {
    //Do something
}
```

Is translated by IntelliJ into this Scala piece of code:

```
{
  var i: Int
  while(i < array.size) {
    //do something
    i += 1; i - 1
  }
}
```

Obviously, this has the same behaviour than the for loop, but it is illegible and not Scala like. Especially when you can use a Scala for loop, or even sometimes a foreach over a collection.

Another problem encountered with the automated translation is that it didn't replace Java collections with Scala collections. The easiest way to get the code to work is to use the scala.collection.JavaConversions._ package. However, the best solution remains replacing collections by hand. Java sets and maps are replaced into Scala sets and maps and array lists are replaced by array buffers.

The translation of constructors was a problem too. Indeed in Java the constructors are of the form:

```
public MyClass(args) {
    super();
    [...]
}
```

On the other hand, in Scala the default constructor is defined in the class declaration, but it is possible to add multiple constructors by defining the method this, $def this(args)$.

```
//Correct translation of the Java
    constructor
class MyClass(args) extends SuperClass {
    [...]
}
```

However, IntelliJ didn't translate the default constructor as class definition, but by defining *this*. Furthermore, when the Java constructor calls the super constructor, it is translated into ‘*super*‘(), when in Scala, the call to the super constructor is made in the class declaration. There is no such thing as super().

```
//ItelliJ translation of the Java
    constructor
def this(args) {
  ‘super‘()
  [...]
}
```

Finally, the automated translation put all the field at the end of the class. As the body of the class is also the constructor, most of the fields values that have been set in the class body are replaced at the end of the class by the fields initial values. In addition, the initial values are often set to null, which leads to a lot of null pointer exceptions.

3.3 Code modifications

Now that we have obtained a first compiling version of SHotDraw, we can start to change the code. The first step is to fix all the runtime errors and bugs created by the automated translation. Indeed, it is preferable to have a bug free version to start changing the code, otherwise we would not know if an error was there before or is the result of a modification.

As the code comes from Java, many fields are initialized to null. Instead of using null as initial value, we could use Scala options. However, it would require lots of modifications and create many objects, using a lot of memory. Another way to replace the null initial value is to create null objects. One example is the NullTool, extending AbstractTool with empty implementations for methods. Similarly we created objects NoDragSource, NoDropTarget, NoDragGestureRecognizer and NoImage to replace options in DNDHelper and in Iconkit.

In JHotDraw, the figure attributes are managed in a single map, Map[name: String, attribute: Any]. The good point is that it is scalable, you can indeed put what you want in that map. On the other hand, the fact that the attribute is of type Any, requires either a cast or a pattern matching when retrieving the value. In addition, there are not that much attributes: *bordercolor*, *fillcolor*, *fontsize*, *style*, *nameandcolor* and *arrowstyle*. As it is not very useful to have something very scalable in that case, we chose to replace the map with fields, with getters and setters.

```
trait FigureAttributes extends
    Serializable {
  import PolyLineFigure._
  [...]
  private var _fontName = "Helvetica"
  private var _fontSize = 12
  private var _fontStyle = Font.PLAIN
  [...]
  def fontName: String = _fontName
  def fontSize: Int = _fontSize
  def fontStyle: Int = _fontStyle
  [...]
  def fontName_=(value: String) {
    _fontName = value
  }
  def fontSize_=(value: Int) {
    _fontSize = value
  }
  def fontStyle_=(value: Int) {
    _fontStyle = value
  }
  [...]
}
```

This change in the way attributes are handled lead to a change in the way the change attribute command works. Indeed every item in a menu is a command and the item in the attributes menu are change attribute commands. The first change was to make the attribute parameter generic. Before the attribute values were Any, now they differ depending of the attribute and we don't want to have one change attribute command for each attribute. The ChangeAttributeCommand must then be generic.

The change attribute command takes as parameter a figure attribute constant depending on what attribute we want to set. The trait FigureAttributeConstant is defined as follows:

```
trait FigureAttributeConstant[T] {
  def setAttribute(figAttr:
      FigureAttributes, value: T)
  def getAttribute(figAttr:
      FigureAttributes): T
}
```

And as an example of implementation, the declaration of the frame color attribute constant,

```
case object FrameColor extends
    FigureAttributeConstant[Color] {
  def setAttribute(figAttr:
      FigureAttributes, value: Color) {
    figAttr.frameColor = value
  }
  def getAttribute(figAttr:
      FigureAttributes): Color =
      figAttr.frameColor
}
```

This solution is better than the first one in the sense that it doesn't require any cast or pattern matching and we won't be able to put a String where we want a color. Nevertheless it is not scalable. We could make it scalable by using virtual classes for instance, but in that case it is not really necessary.

Another big change made to the framework is the creation of an abstract class RectangularFigure. In JHotDraw, every figure defined by a rectangle display box have to implement the display box, the handles and the move method, even though these methods are exactly the same, a lot of useless code duplication. The RectangularFigure class now contains all these methods and the figures only define the way they must be drawn, in code it means that they have to implement the *path* method that returns a GeneralPath, and the way an arrow can connect to them.

In Java it is mandatory to explicitly specify the type at the variable declaration. In Scala however it is sometimes possible not to specify the type, which will be implicitly found thanks to the type inference. Because of the automated translation, the types were always explicitly declared. When possible, it is preferable to have $valtracker = newMediaTracker(this)$ instead of $valtracker : MediaTracker = newMediaTracker(this)$.

The general pattern followed by most of the Scala users is to be to put parenthesis for methods where the side effect is important. This is usually true for all methods returning Unit. For example, in ConnectionFigure, there are methods *updateConnection* or *disconnectEnd* that should read: *updateConnection*() or *disconnectEnd*(). During the translation into Scala, these parenthesis were removed, so we have to add them again.

We didn't only clean the code, we made some improvements too. For instance, in JHotDraw, a new window didn't start with all the tools, but only the selection tool. There is no reason to that, so we added all the creation tools to every window. Another improvement we made was to make the figure creation undoable. Indeed in JHotDraw, every action is undoable except the figure creation, which is sad. Finally, in JHotDraw, to end the creation of a polygon figure, one had to select another tool, releasing the mouse was not sufficient, resulting in very weird shapes. The creation now ends when the mouse is released.

Finally the biggest change made to the code was to add constraints. The rectangular figures and the handles associated to them are now defined by CVar instead of usual variables. From there, we have to change the way moving and resizing are performed, but we can also easily add new features like alignments. These changes are detailed in section Using constraints and section Alignment.

4 Handling multiple inputs

As long as we have only one input corresponding to one action, for instance I can only move a figure by dragging it, everything is fine. Now what if I have multiple inputs triggering the same event ? Let say that I can not only move a figure by dragging it, but I can move it with the arrow keys too. How are the events handled ? Do we allow the figure to be moved with the arrows while being dragged ? This is not the only example where applications have to deal with two or more different inputs, as shown in this non exhaustive list.

- Moving a figure
  - Mouse
  - External device (Kinect, Touch Screen)
  - Keyboard
  - Moving another figure
- Arrow connection
  - Mouse
  - Figure move
  - Keyboard (arrow keys or similar)
  - External device (Kinect, Touch Screen)
- Changing attribute
  - Menu
  - Mouse
  - Keyboard

Another curious behaviour can be observed during the undo operation. Let's proceed as follows, we create a new figure, we then move it to the right and finally we move it back to its initial position. However, before we release the mouse, which means before we end the moving operation, we hit $ctrl + z$. Depending on the application, the behaviour may vary. In JHotDraw, the undo operation is instantly executed, thus the figure is moved twice to the left and goes out of the screen. Some application chose to buffer the undo operations and execute them after the moving operation ended. Finally, some other application just ignore the undo operation during the move.

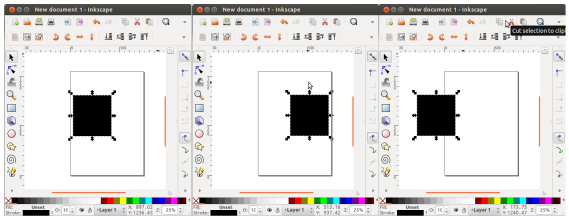

Figure 2: Inkscape not handling correctly a undo operation during a move.

What is the correct way to deal with this particular case ? Should we buffer the undo operations and perform them afterwards, should we interrupt the move operation to perform the undo or should we simply disable the undo operation during a move ? Obviously the solution offered by JHotDraw, that is performing both at the same time, is not acceptable. But it is not the only application with such issues. Indeed, we can find similar errors in multiple well known interactive applications. Inkscape for instance, allows the undo operation during a move and thus the figure may move twice and end up out of the editor pane, as shwon in Figure 2. On the left, we can see the rectangle initial position. We then move the figure to the right. Finally, we move the rectangle back to its inital

position, but we hit ctrl+z before we release the mouse. The rightmost image shows where the figure ends up: out of the editor area. Autodesk Sketchbook, Google docs, Photoshop express editor or LibreOffice are other examples of interactive applications with problems when handling with multiple inputs.

5 Removing observers

Now that we have a working version of SHotDraw, we can start removing observers. There are multiple reason why we want to remove observers. For instance, an observer tracking a mouse input may call a drawing method when such event occurs, hence the separation of concerns is violated. For the same reasons, it is not easy to add or remove an observer. These are reasons why we would like to have another way to handle inputs and events.

5.1 Scala.React

One way to replace the observers is Scala.React. Scala.React is a library of programming abstractions allowing the user to migrate from observers to a reactive data-flow model. The framework is based on the concepts of *reactors*, *events* and *signals*.

The Scala.React framework eventually address all of the issues generated by the use of observers. Indeed, for instance, you define a reactor at a single place. That way the separation of concern is not violated and it is very easy to add or remove a reactor. Furthermore, with Scala.React and the concept of reactors, we can turn an observer into a code without inversion of control.

5.2 Constraints

Another option to replace observers is to use a constraint system solver. We aim to replace the interaction between different elements by defining constraints. Thus we only have to add the constraints where we need them and leave the resolution to the solver. This way, it is very easy to define the interaction between the different elements of a drawing.

The constraint system solver is very useful when we have dependencies in both directions. For instance, the handles of a rectangle need to move with the rectangle. On the other hand, the rectangle needs to be resized when the one handle is dragged. Using constraints to define the position of handles relative to the rectangle seems to be the logical solution. The solver will compute the rest. However, it is hardly possible to handle events with the constraints system.

6 Using constraints

As seen above, one way to remove observers is to add constraints. For instance, to set the position of the top left handle relative to a rectangle, we can add two constraints: $handle.x == rectangle.x$ and $handle.y == rectangle.y$ and when we move either the rectangle or the handle, we will ask the solver to solve the system. There exist a lot of algorithms to solve constraints systems, mostly used in artificial intelligence or graph theory. The two main methods to solve a constraint system are variable elimination, like the Gaussian elimination algorithm, and local consistency, like the path consistency or the arc consistency algorithms.

The variable elimination algorithms only allow constraint under the form of equations or will try to transform other constraints into equa-

tions. From then, the goal is to obtain a normal form for the equation. The normal form is an equivalent formulation of the constraints where a variable appears only once in the left hand side of an equation. The variable elimination algorithm computes the normal form by replacing multiple occurrences of variables.

The local consistency algorithm on the other hand divides the initial problem into small overlapping sub-problems. The algorithm repeatedly tries to simplify the sub-problems and propagate obtained constraints, until a fixpoint is reached. As the size of the sub-problems is fixed, there exist only a polynomial number of possible sub-problems and, if we can solve the sub-problems in a polynomial time, we can hope to solve the full problem in polynomial time.

6.1 Cassowary

Cassowary is an incremental constraint satisfaction algorithm that can solve systems of linear equality inequality constraints. Such constraints are very useful in a graphical interface to specify the position of one element relative to another. The best example is the position of the handles relative to the figure they are attached to. Using equality constraints allows to specify the position of one handle relative to the position, width and height of the figure. Furthermore equality and inequality constraints are very useful to determine the position of two figures relative to the other, for instance to express the requirement that one figure should be left to the other.

Cassowary uses the simplex algorithm to solve the constraints system. However, the simplex algorithm doesn't handle variables that can have negative values. Nevertheless it is possible to use the augmented simplex form to deal with unrestricted variables (variables that can have negative values). The idea is to use two tables instead of one. In one table, we put all the unrestricted variables and the other variables, the one restricted to be non-negative, in the other.

The simplex algorithm is used to compute an optimal solution for the table of restricted variables. We can then determine the values of the unrestricted variables using the second table. The first step of the algorithm is then to separate the constraints into the two tables. At first, we need to transform the inequalities into equations, using slack variables. For instance, the inequality $x \leq 42$ can be transformed into $x + s = 42 \wedge s \geq 0$, where s is the slack variable. When we have turned all the inequalities into equations, we can separate the variables into two tables. In the beginning, all the equations are in the restricted table. Then, until there is no more unrestricted variables in the restricted table, we move an equation with an unrestricted variable to the other table. We then solve the moved equation, for instance $x+s = 42$ becomes $x = 42 - s$, and substitute the result for all occurrences of the variable. We can now use the simplex algorithm to compute an optimal solution.

The advantage of Cassowary is that it solves the constraint system more efficiently, hence it is very useful for interactive applications. Indeed we want the elements to move fast, solving a constraint system may require some time and introduce a latency. Another advantage is that it is possible to specify a strength on the constraints. Required means the constraint must be satisfied, but the strong and weak strengths are only labels allowing the user to add preferences. On the other hand the Cassowary library offers no documentation and the errors are not very clear, making the implementation and especially the debugging difficult.

6.2 Updating SHotDraw to use constraints

To change the value of a variable affected by some constraints, the simplex solver needs to be able to modify the value of other variables affected by the same constraints. Indeed, the constraints can not be set on normal variables. This will require two changes, one in the variables declaration and one in the way we change the value of a variable. First, the solver needs the variables to be mutable, that is to use CVar instead of normal variables. Then, when changing the value of one variable, we can not say $value = newValue$, but we have to follow the following steps:

```
/* add the variable which
 * value will to change */
solver.addEditVar(cvar).beginEdit

/* Suggest a new value and
 * solve the system */
solver.suggestValue(cvar, value).resolve

/* And finally notify the
 * end of edition */
solver.endEdit
```

In order to use constraints with the figures, we will need to change a little bit the way figures are defined. We will only use constraints with figures that are represented by a rectangle display box and we create a new subclass $RectangularFigure$ of the $AbstractFigure$ to represent these figures. A rectangular figure is represented by four CVar, the top left corner, the width and the height of its display box. We need to redefine the move operation, as we now need to follow the edit steps mentioned above.

We will also need to change the way handles are define. Any rectangular figure uses the eight new handles, defined by the class $DraggableBox$. A draggable box is defined by its center position as CVar and its width and height. And similarly to the rectangular figure implementation, we need to redefine the drag operation to follow the edit steps. With this new implementation we are ready to add constraints.

6.3 Resize dialog

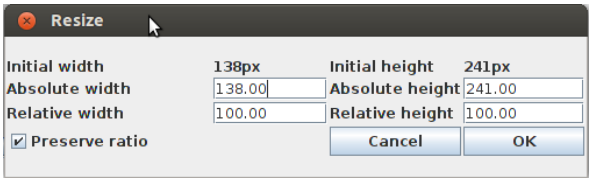

Figure 3: The resize dialog

An easy way to add constraints in SHotDraw is to create a resize dialog. The reisze dialog is composed of 4 widgets containing the absolute width and height in pixels and the relative width and height in percentage of the initial value. If the user wants to modify one of these values, for instance the absolute width, then the relative width should be modified as well. And this is were we can use constraints. Indeed, we want that $absolutewidth == (relativewidth * initialwidth)/100.0$. The equation is the same for the absolute and relative height and we end up with two constraints:

```
absW :==relW*initW/100
absH :== relH*initH/100
```

We may want to preserve the ratio between height and weight as well. There are multiple ways to create a ratio constraint, but the easiest way is to ensure that the relative height and the relative width are equal. Indeed, in

the beginning, whatever the absolute height and width values, the relative height and width will be 100%.

```
relW :== relH
```

Now that we have this new constraint set, we will need to enable or disable it. This is why the resize dialog contains a ratio checkbox and the ratio constraint is stored in a variable, to keep a trace of its address, hence being able to add or remove it to the solver. Finally, the set of constraints for the resize dialog is:

```
val ratio = relW :== relH
ensure(absW :== relW*initW/100)
ensure(absH :== relH*initH/100)
ensure(ratio)
```

If we had to implement this resize dialog without using a constraint system solver, we would have to notify every field that its value should be updated. This is a lot heavier when we can just add three constraints and leave the rest to the solver.

6.4 Handles

A more complex but more powerful use of constraints is to set the position of the handles relative to the figure they are attached to. Indeed, if we move the figure, we want the handles to move too and conversely, if we move one handle, we want the figure to be resized. This means we would have to use bidirectional notifications or add constraints between the figure position and the handles. As it is easier to set constraints and leave the work to the simplex solver, we of course choose the constraints solution.

The figure display box is defined by its top left corner and its width and height and the handles are defined by their center. From this, we can set the position of every handle relative to the figure position, width and height.

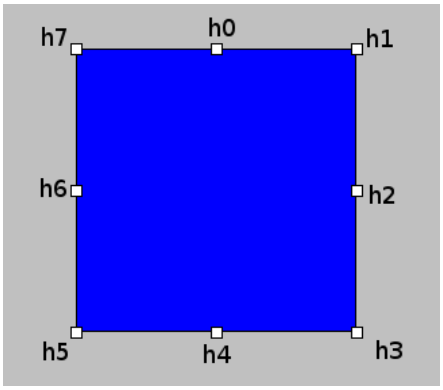

Figure 4: The handles of a figure and the numbering used to access them.

Horizontally, we have that $h5, h6, h7$ have the same x value, which corresponds to the display box x value. We know that $h0$ and $h4$ are equal to $x+width/2$ and finally, we have $h1 == h2 == h3 == (x + width)$. Similarly, we can compute the y values relative to the figure and it goes as follow. $h0, h1$ and $h7$ are equal to y, $h2$ and $h6$ correspond to $y + height/2$ and $h3 == h4 == h5 == (y + height)$.

Finally, we end up with the following set of constraints:

```
//db stands for displaybox
//Horizontally
h(5).x :== db.x
h(6).x :== h(5).x
h(7).x :== h(5).x

h(0).x :== db.x+db.width/2
h(4).x :== h(0).cx

h(1).x :== db.x+db.width
h(2).x :== h(1).x
h(3).x :== h(1).x

//Vertically
h(0).y :== db.y
h(1).y :== h(0).y
h(7).y :== h(0).y

h(2).y :== db.y+db.height/2
h(6).y :== h(2).y

h(3).y :== db.y+db.height
h(4).y :== h(3).y
h(5).y :== h(3).y
```

In order to use constraints on the handles, we have to change the implementation. To do so, we create a new class called DraggableBox to represent handles which position is defined as CVar instead of normal variables. We have to change the way the drag operation is performed too. During the mouse down event, we want to add constraints and add edit vars depending on the position of the handle. The corner handles will perform a resizement in both direction, so we add x and y as edit vars. But if we move horizontally or vertically, we will only add x, respectively y. Furthermore, when moving in both direction, that is dragging a corner handle, we want the opposite corner to stay. When moving horizontally, we want the opposite handle to stand as well, but we also want to disable any vertical movement. This is why we want at least one of the top handles and one of the bottom handles to stay. The constraints are very similar for a vertical move.

```
override def invokeStart(x: Int, y: Int,
    view: DrawingView) {
  solver.resetStayConstants()
  direction match {
    case DraggableBox.North =>
        owner.northMove()
    case DraggableBox.NorthEast =>
        owner.northEastMove()
    case DraggableBox.East =>
        owner.eastMove()
    case DraggableBox.SouthEast =>
        owner.southEastMove()
    case DraggableBox.South =>
        owner.southMove()
    case DraggableBox.SouthWest =>
        owner.southWestMove()
    case DraggableBox.West =>
        owner.westMove()
    case DraggableBox.NorthWest =>
        owner.northWestMove()
  }
  direction match {
    case _:DraggableBox.Both =>
        solver.addEditVar(cx).addEditVar(cy).beginEdit
    case _:DraggableBox.Horizontal =>
        solver.addEditVar(cx).beginEdit
    case _:DraggableBox.Vertical =>
        solver.addEditVar(cy).beginEdit
  }
}
```

During the drag operation, we want to suggest values for the edit variables and resolve the constraint system. To have a fluid resizement, we can refresh the figure. Once again, we have to

take care of the direction. Indeed, if we suggest a value for a variable that is not an edit variable, we will have an error.

```
override def invokeStep(x: Int, y: Int,
    anchorX: Int, anchorY: Int, view:
    DrawingView) {
  direction match {
    case _:DraggableBox.Both =>
        solver.suggestValue(cx,
        x).suggestValue(cy, y).resolve
    case _:DraggableBox.Horizontal =>
        solver.suggestValue(cx, x).resolve
    case _:DraggableBox.Vertical =>
        solver.suggestValue(cy, y).resolve
  }
  owner.changed()
}
```

Finally, when the mouse is released, we will end the edition, remove the constraints we added during the operation and of course refresh the figure.

```
override def invokeEnd(x: Int, y: Int,
    anchorX: Int, anchorY: Int, view:
    DrawingView) {
  solver.endEdit
  solver.resetStayConstants()
  owner.changed()
}
```

7 Alignment

Now that we have a constraint system in place, we can do more. Alignment is a very good example of the powerfulness of constraints. Let's say we want to align two rectangles on the left side of one of them, we just have to add a constraint saying $rectangle1.x :== rectangle2.x$. There no easiest way. However, the model would have to change a little bit. Indeed, when using constraints only in one figure, that is no constraints defined on variables from two or more different figures, we could define one solver for each figure. Now that we have constraints involving multiple figures, we need to have one and only one solver for the entire application. Indeed, moving one figure will not only affect its handles, but it may affect another figure, hence the figure's handles too.

7.1 Implementation

We want to be able to define as many alignment as desired and not only a horizontal alignment between two figures. The idea is then to have a alignment framework that allows the user to define many different ways to align figures, like left or right alignment, top or bottom, or even align figures on their width and height.

7.1.1 Align class

The structure of the alignments is defined in the Align abstract class. Basically, it is only a set of constraints that we want to add to the solver, so the only thing subclasses will need to provide is the set of constraints that correspond to some alignment, $rectangle_i.x :== rectangle_j.x$ for a left alignment for instance.

```
abstract class Align (name: String, view:
    DrawingView) {

  private var enabled_ = false
  private var consts: List[Constraint] =
      List()

  def constraints: List[Constraint]
  def enabled = enabled_

  def enable() {
    if(consts isEmpty)
      consts = constraints

    consts foreach { c =>
      view.solver.addConstraint(c)
    }
    enabled_ = true
  }

  def disable() {
   consts foreach { c =>
     view.solver.removeConstraint(c)
   }
   enabled_ = false
  }

  override def toString = name
}
```

The Align class provides the enable and disable methods, that will enable or disable the alignment by adding or removing all the constraints from the solver. The class also provides a enabled flag, which is used by the align manager to know if a constraint is enabled or not, so that we can show which constraints are enabled and which are not. In the end, the only part that is defined by the user is the way constraints are created. They have indeed to implement the abstract method constraints, which result will be stored by the superclass, the first time the alignment is enabled.

7.1.2 Align manager

The align manager is the place where all the existing alignments are stored, enabled or not. Having all the alignments stored in one place is very useful to show them to the user. We can then retrieve the list of all alignment and offer the opportunity to the user to enable or disable one of them or all at the same time.

7.1.3 Integration to the application

To integrate a new alignment to the application, we need to create a new menu item. To follow the framework, we want to add the alignment items the same way it is done for other menu, like attributes. The new menu item we add must extend the AbstractCommand class, which represent a command that can be executed. This is why we create a new AlignCommand class, that will implement the execute method and define the way the command handles undo and redo operations. When the command is executed, we create a new instance of the alignment with the selected figures, using the AlignFactory given as parameter. We then add this new alignment to the manager and finally we enable the alignment.

7.2 Examples

As an example, let's consider the left align we talked about above. The only thing the LeftAlign class needs to do is to define the set of constraints on the affected figures. In this example, the set of constraint is created this way: $\forall i, j \in \mathit{selected\ figures} \wedge i \neq j : \mathit{add}(i.x :== j.x)$.

Finally the LeftAlign class looks like this.

```
class LeftAlign(view: DrawingView) extends
    Align("Left", view) {

  override def constraints =
      view.selection.
      foldLeft(List[Constraint]())((l,f) =>
      f match {
    case rf: RectangularFigure => l :::
      view.selection.
          foldLeft(List[Constraint]())((l,f)
          => f match {
        case rff: RectangularFigure if rff
            != rf => (rf.db.cx :==
            rff.db.cx) :: l
        case _ => l
      })
    case _ => l
  })
}
```

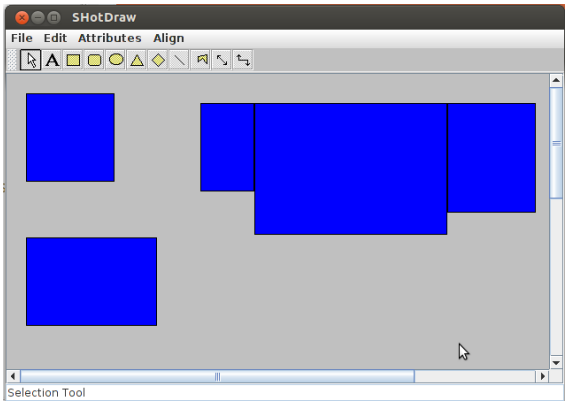

Figure 5: On the left, two rectangles affected by a left alignment. On the right, three figures affected by a side by side alignment coupled with a top alignment.

Another example of alignment is the side by side alignment. The constraints method is bit more complex, because we have to sort the selected figure from left to right and then add a constraint between the right side of one figure and the left of the next figure. As said before, a rectangular figure is defined by its display box which is defined by its top left corner, width and height. Thus we can define the set of constraints as $figure_i.x + figure_i.width :== figure_j.x$ where $j = i + 1$ and in code:

```
override def constraints = {
  var l = List[Constraint]()
  var last: RectangularFigure = null
  view.selection.sortWith((f1,f2) =>
      f1.displayBox.y <
      f2.displayBox.y).foreach ( f => f
      match {
    case rf: RectangularFigure =>
      if(last != null) l ::= (rf.db.cy :==
          last.db.cy+last.db.cheight)
      last = rf
    case _ =>
  })
  l
}
```

8 Conclusion

In conclusion, this project was very interesting and very instructive in many way. First of all, al the translation process made me write a lot of Scala and, more important, compare the Scala code with the Java code. That way, I learned a lot about Scala mechanics and style and why and when it is better than Java. I particularly like the way you can perform actions on collections.

More than improving my understanding of Scala, this project gave me the occasion of learning how an interactive application works, what are the mechanisms behind what we see and that an application that may seem simple actually re-

quires a lot of engineering. Translating the full code into Scala was very helpful to understand the operation of HotDraw.

Finally, I learned a lot about how we can change the way we handle events. Using constraints instead of observers is very ingenious. It just blew my mind how easy we can add new features, like alignment, on a framework that uses constraints. You can just define how the different elements interact with each other and let the solver do the job. However, solving the whole system every time one figure moves is not very efficient and we can see a little latency compared to the version without constraints. It is sad I didn't have the time to play with Scala.React.

References

[1] Ingo Maier, Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. 2012.

[2] JHotDraw

[3] Thomas Fruehwirth, Slim Abdennadher. *Principles of Constraint Systems and Constraint Solvers*. 2005.

[4] Greg J. Badros, Alan Borning, Peter J. Stuckey. *The Cassowary Linear Arithmetic Constraint Solving Algorithm*. 2002.

[5] IntelliJ

[6] Jaakko Jaervi, Mat Marcus, Sean Parent, John Freeman, Jacob N. Smith. *Property Models. From Incidental Algorithms to Reusable Components*. 2008.

[7] SHotDraw