

Diagram Editor for Tablet PC

Term project for Brainchild

PROJECT REPORT

Project period: July 2006 – November 2006
Student name: Ruihua Jin
Status: 8-th Semester
Email address: rjin@student.ethz.ch
Supervisor name: Christoph Angerer

Introduction

With traditional diagram editors the user must first select the type of the shape he wants to draw, and then use the mouse to draw the shape, and each time before the user performs some operation on a certain shape, he must select the corresponding menu item. This approach does not take advantage of a Tablet PC with which one can draw strokes on the screen as if one were drawing on a piece of paper. Tablet PC promises more comfort in freehand drawing, and at the same time, a Tablet PC also offers the possibility to perform operations on strokes or shapes like a common computer, which a piece of paper is not able to do.

Using ink recognition technique the user overcomes the shortcomings of freehand drawing where straight lines and regular geometrical shapes are almost impossible to draw; and using gesture recognition technique the user can call a command simply by drawing some stroke without having to search for certain menu items to get the command performed. Tablet PC is therefore an ideal presentation tool for meetings, lectures, etc.

The current project is one approach taken to offer the user a comfortable way to draw free strokes, geometrical shapes and write texts; Furthermore, it also offers several gestures for the commands which are commonly used in a diagram editor.

Overview

The goal of this project is to develop a diagram editor for Tablet PC which allows the user to use the pen to draw diagrams.

Scope of the work

Using this diagram editor the user can draw arbitrary figures, regular geometrical shapes and write texts. Corresponding to these three functions there are three drawing modes supported: freehand style, that is, what you draw is what you see, no ink recognition will be involved, and non-freehand style, that is, the figure drawn by the user will be first classified as a regular geometrical shape, e.g. a line, an arrow, a rectangle, etc., then the corresponding geometrical shape will be output on the screen, i.e.:



at the same location. Freehand style is the default drawing mode. The user can switch between these three modes by clicking the corresponding button in the pie-menu:

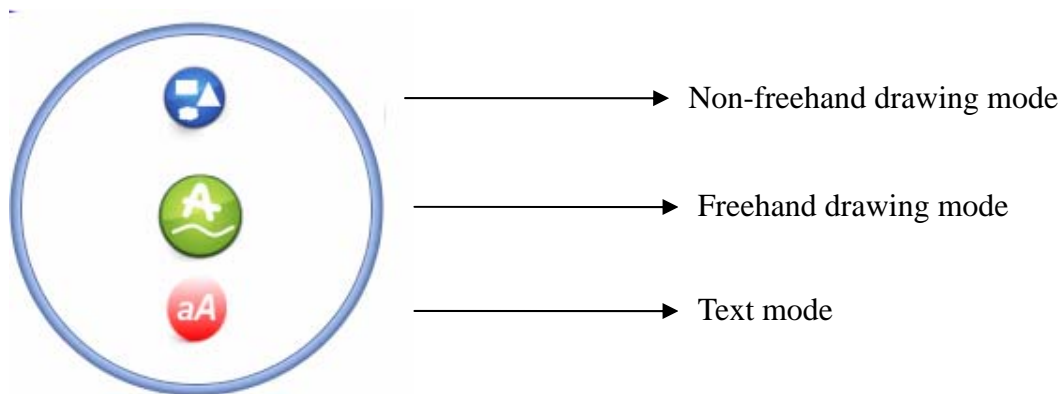


Figure 1. Pie-menu for selecting drawing modes.

The current project concerns only the non-freehand drawing mode and the text mode.

In non-freehand drawing mode the user can draw basic geometrical shapes which are drawn with only one stroke, complex geometrical shapes which are drawn with more than one strokes one after another, and flow graphs in which only a restricted number of shapes can be drawn. When non-freehand drawing mode is activated, the following pie-menu pops up in the GUI, where “Basic” is the default (flow graph mode is not implemented in the current project):

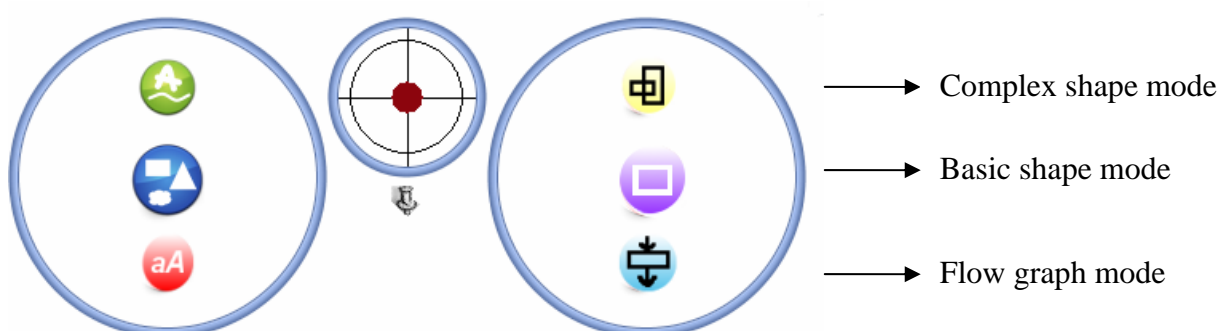


Figure 2. Pie-menu when non-freehand drawing mode is activated.

When the text mode is activated, four circular controls will be shown at the bottom of the screen, the user can then write letters, numbers or punctuations one by one in theses controls, they will then be recognized and output in the wanted text area.

I. Basic shape mode: Recognizing and constructing basic geometrical shapes

With basic geometrical shapes the author means the geometrical shapes which can be drawn with a single stroke. Following shapes are supported as basic geometrical shapes:

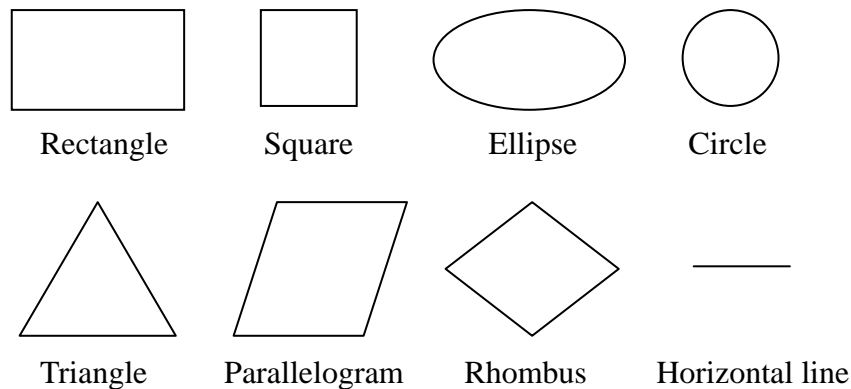


Figure 3. Supported basic geometrical shapes in basic shape mode.

The recognition is based on Rubine's Recognizer, which requires one to follow certain rules while drawing shapes. To it, which end one starts at and the direction one draws in are important. For example, the recognizer would treat a "1" drawn top-to-bottom and a "1" drawn bottom-to-top differently. Here, shapes are always drawn starting at the dot and it is also important that the user draws shapes in the correct direction:

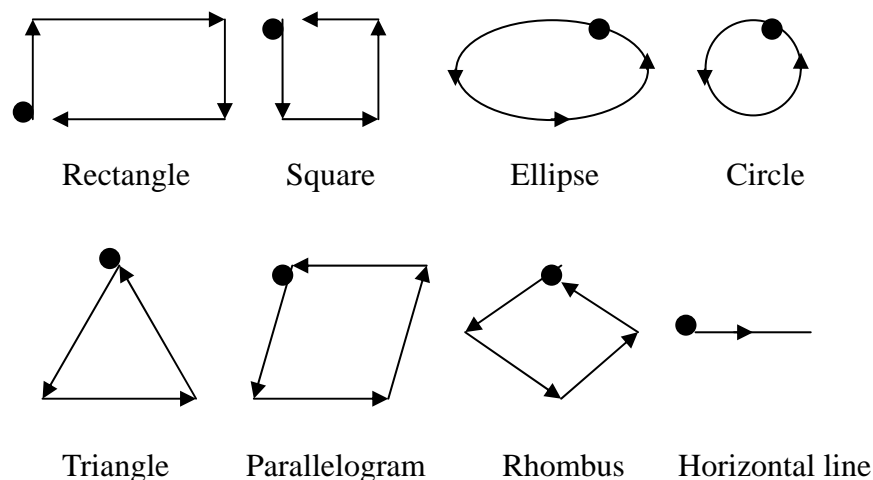


Figure 4. Ways to draw basic geometrical shapes.
The dot signifies where one starts drawing, the arrow signifies the direction in which one draws.

Arrows are also supported as basic geometrical shapes:

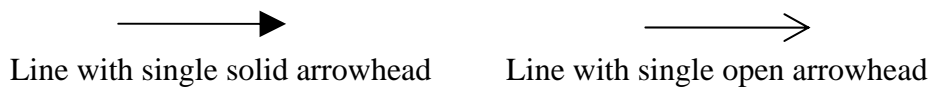


Figure 5. Supported arrow shapes in basic shape mode.

where each shape should be drawn as the following:

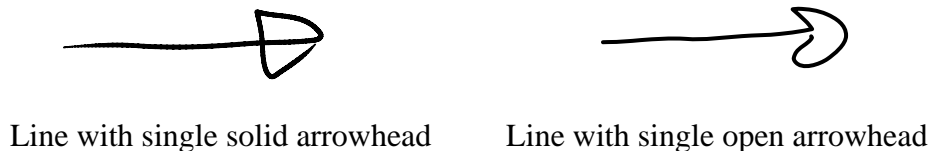


Figure 6. Rule to draw arrow shapes in Figure 5.

The drawbacks of using Rubine's recognizer to recognize basic geometrical shapes are the following:

- The user must follow predefined ways to draw shapes.
- The shapes cannot always be recognized correctly. Sometimes the recognizer returns no result, since the probabilities of all the classifications are under the threshold (default is 0.8); and sometimes the shape is misclassified, i.e. ellipses are often classified as parallelograms. Reason: the user cannot always draw shapes perfectly and the recognizer is not able to classify shapes with 100% certainty as long as it only refers to a small number of gestures it get feeded previously.
- In order to enable the recognizer to recognize shapes in arbitrary size and in arbitrary direction, one should offer the recognizer a file which contains the corresponding shapes in arbitrary size and in arbitrary direction, which is a tedious work and may even lead to worse classification results. So in basic shape mode which uses Rubine's recognizer the author only takes care of the "horizontal" geometrical shapes, if the user wants to draw geometrical shapes in arbitrary size and in arbitrary direction, then he should switch to the complex shape mode which is considered to overcome these shortcomings of a recognizer.

If none of the above shapes can be recognized, then the program assumes that the user wants to draw a line, and a line is drawn from the start point to the end point of the stroke, under the condition that the distance between the start point and the end point is larger than 20 pixels, otherwise nothing is drawn. In this way, lines of arbitrary length and in arbitrary direction are supported in basic shape mode.

Since arrows are similar to lines, one only needs to give a small hint to the program to get an arrow of the length and in the direction one wants. First the user draws a line of the intended length and in the intended direction, then he draws one of the following gestures, using right mouse button, on the line:

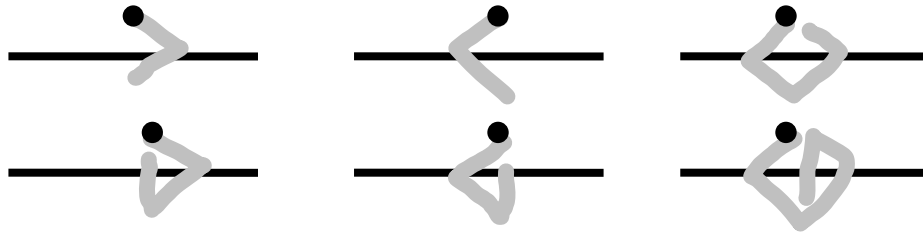


Figure 7. Ways to change a line into an arrow.
The dot signifies where one starts drawing.

The results are

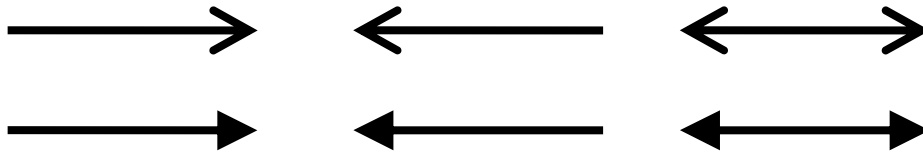


Figure 8. Outputs after drawing gestures shown in Figure 7 on lines.

In sum, in basic shape mode one can draw rectangles, squares, triangles, parallelograms and rhombuses, one can also draw ellipses and circles, however, the sizes of these shapes should not be too large and should also not be too small, and the direction must be horizontal. On the other hand, lines and arrows can be of arbitrary length and in arbitrary direction.

To draw rectangles, squares, triangles, parallelograms and rhombuses of arbitrary size and in arbitrary direction one should switch to complex shape mode.

II. Complex shape mode: Recognizing and constructing complex geometrical shapes

With complex geometrical shapes the author means the geometrical shapes which are drawn with more than one strokes one after another.

To start drawing a complex geometrical shape, the user first activates the complex shape mode, then he draws a stroke along the left side of the screen, using the right mouse button (the program regards any stroke whose x-coordinate of the start point is less than 15 pixel as a signal to start or finish drawing complex shapes), then the text “drawing” shows up to tell the user that he can begin to draw the shape. In the following phase, each stroke drawn by the user is straightened:

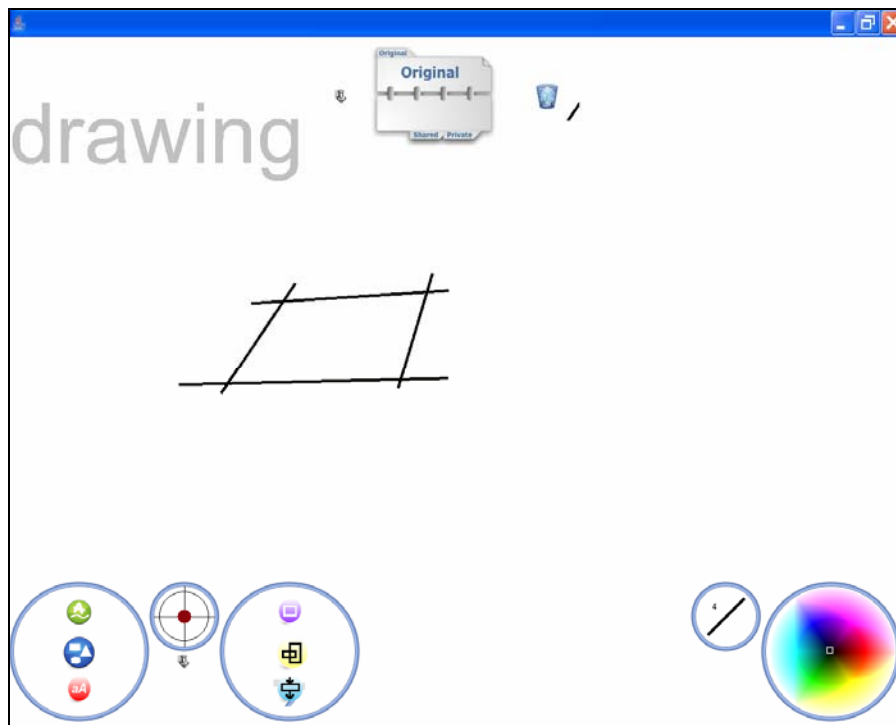


Figure 9. Screen output when a complex shape is being drawn.

When the user finishes drawing the shape, he draws again a stroke along the left side of the screen, using the right mouse button. Right after that the program begins to process the strokes and tries its best to classify the geometrical shape.

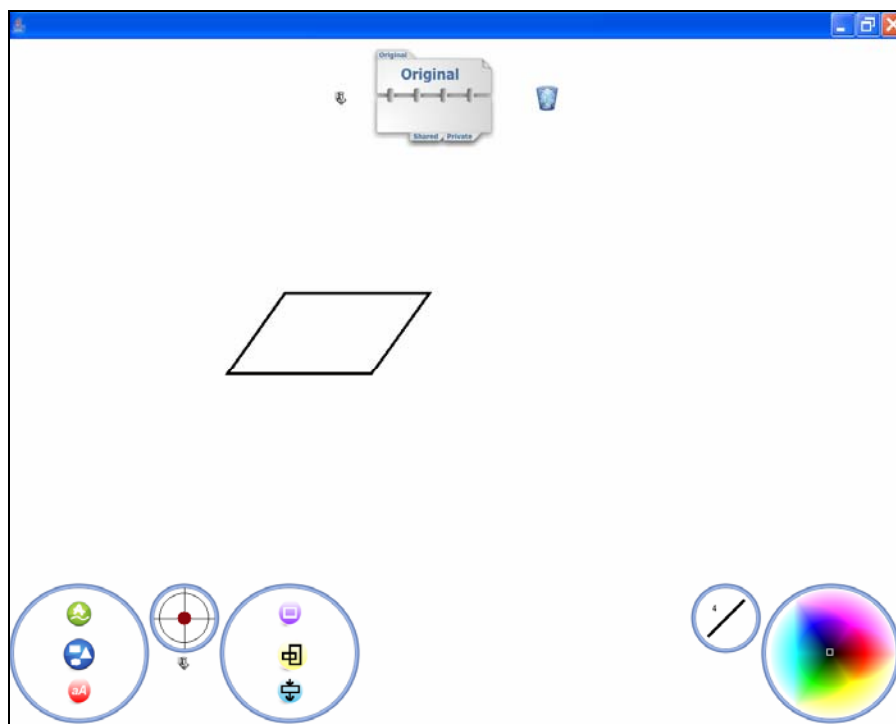


Figure 10. Screen output after the program processes the strokes drawn. In this example, the program classifies the shape as a parallelogram.

During processing the program performs following steps:

1. Start with the first stroke.
2. Merge the first stroke with the second stroke, the second with the third stroke, and so on, until the last stroke.
3. The shape is checked for closedness. The shape is considered as closed if the strokes are drawn either in clockwise order or in counterclockwise order, and the intersection point of the last stroke and the first stroke is contained in the bound of the last stroke or in the bound of the first stroke.
 - 1) If the shape is not closed, then it is drawn as an open path, the processing stops here.
 - 2) If the shape is closed, then the last stroke is merged with the first stroke, and the shape is a closed path. If the number of the strokes is four, then the program checks whether it is a square, a rectangle, a rhombus or a parallelogram, applying the following criteria:
 - 2.1) If the three internal angles are between 70° and 110° , and the differences between the strokes' lengths are less than 25 pixels, then the shape is a square;
 - 2.2) If the three internal angles are between 70° and 110° , then the shape is a rectangle;
 - 2.3) If the sum of two neighbouring internal angles is between 160° and 200° , and the differences between the strokes' lengths are less than 25 pixels, then the shape is a rhombus;
 - 2.4) If the sum of two neighbouring internal angles is between 160° and 200° , then the shape is a parallelogram.

Below are two examples of how to draw a rectangle and a parallelogram in complex shape mode:

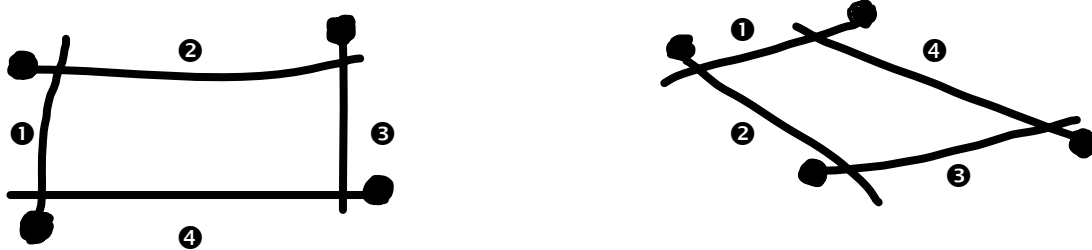


Figure 11. Examples of drawing a rectangle and a parallelogram.
The dot signifies the starting point of a stroke, the numbers signify the order in which the strokes are drawn.

III. Text mode

In text mode, the user first draws a rectangle to specify a text area, using the left mouse button. Then the activated text area is highlighted with a yellow background.

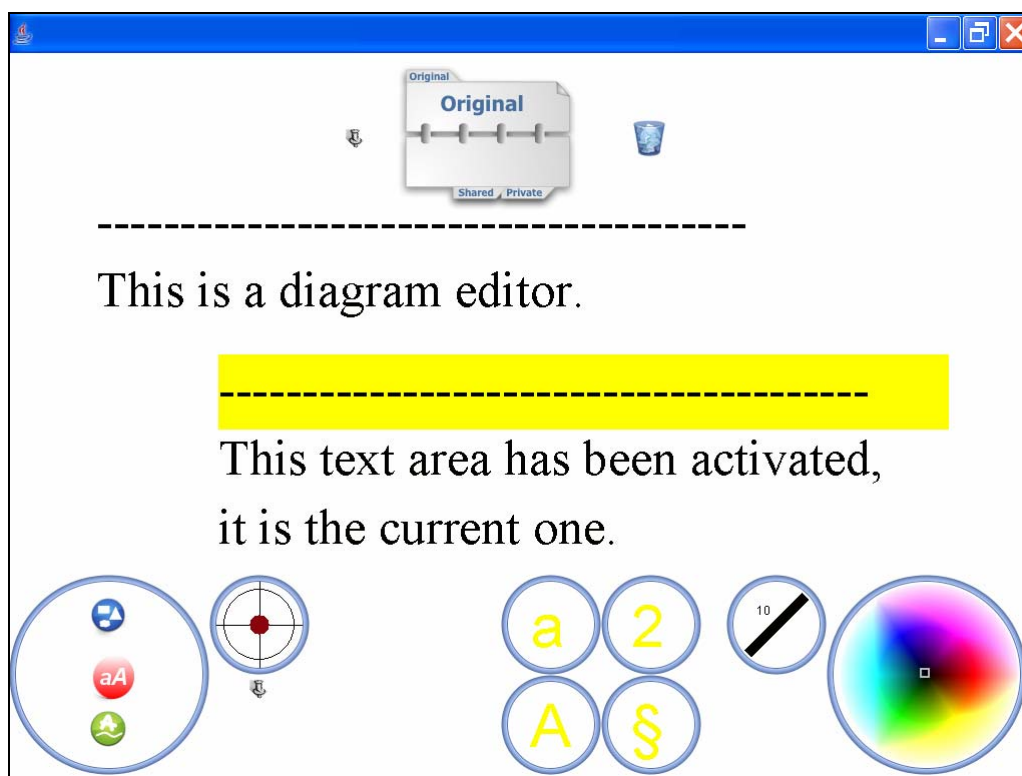


Figure 12. Diagram editor in text mode, with two text areas drawn.

To write letters, numbers or punctuations, the user draws the characters in the four circular controls shown in Figure 12. The one labeled with “a” is for small letters from ‘a’ to ‘z’, the one labeled with “A” is for big letters from ‘A’ to ‘Z’, the one labeled with “2” is for numbers from ‘0’ to ‘9’ and the one labeled with “§” is for punctuations.

In order that the recognizer is able to recognize the characters, the user should follow the following drawing rules:

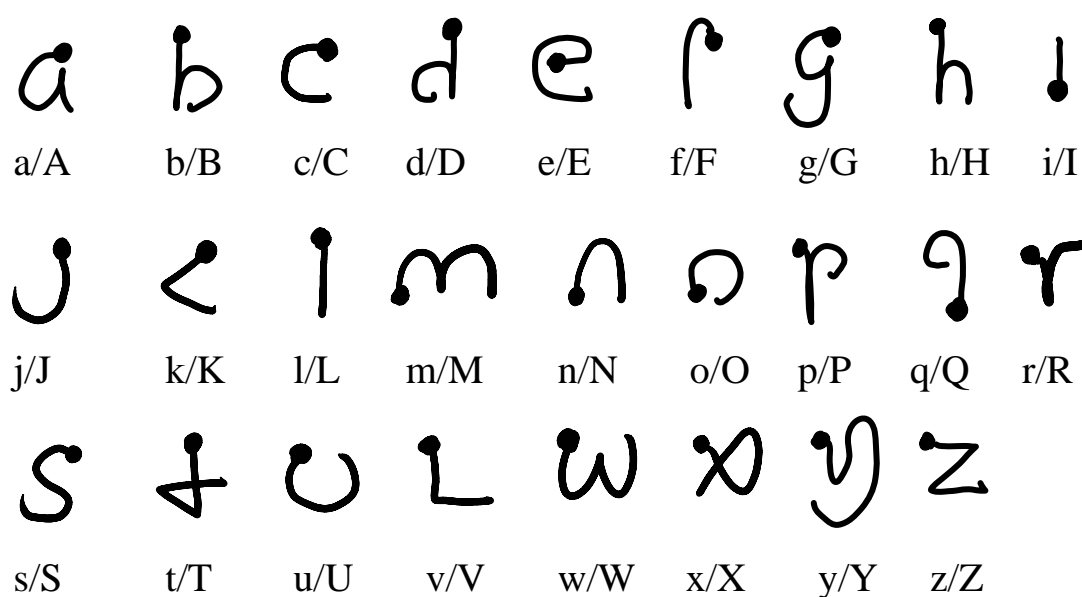


Figure 13. Drawing rule for letters. Heavy dot indicates starting point.

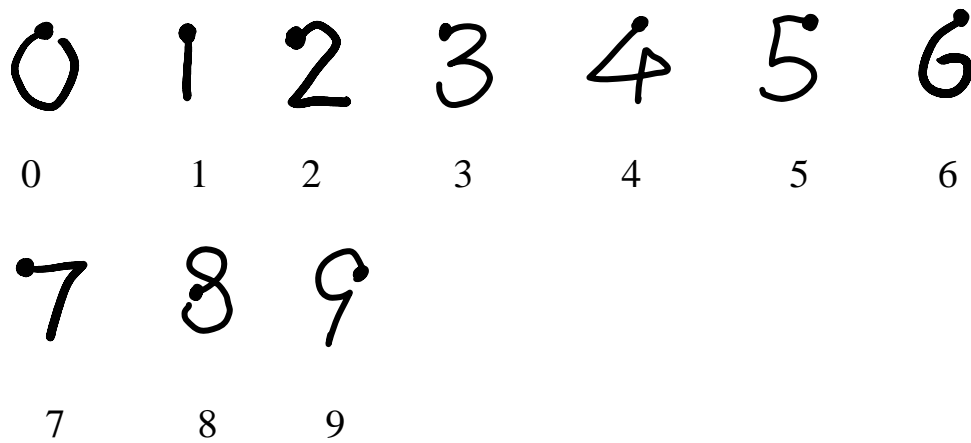


Figure 14. Drawing rule for numbers. Heavy dot indicates starting point.

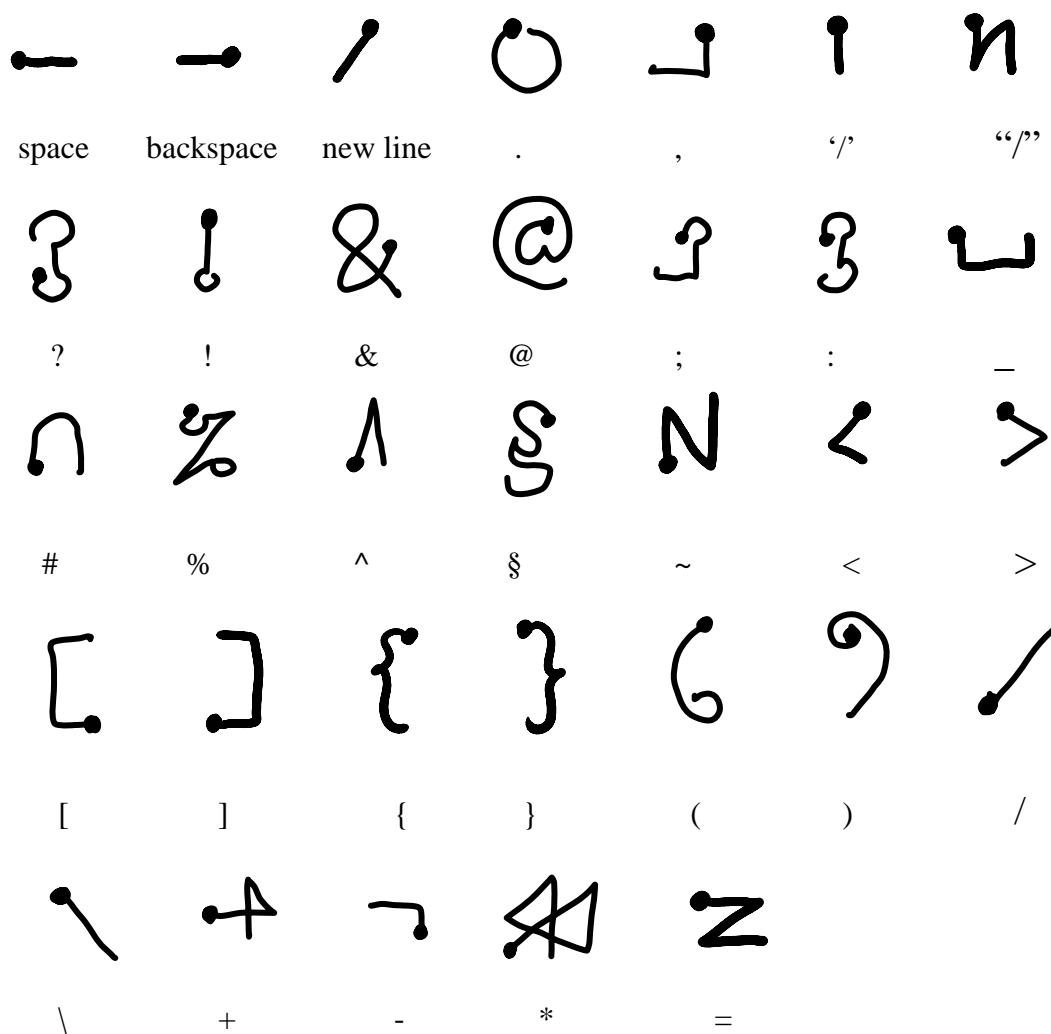


Figure 15. Drawing rule for punctuations. Heavy dot indicates starting point.

To activate a text area, the user left-clicks in the text area, right after that the text area is highlighted with a yellow background, and the following characters will be output in this text area. To deactivate all the text areas, the user just left-clicks outside the text areas.

IV. Context-aware drawing

For the current project, the context-aware drawing only applies to arrows. When an arrow is drawn, the program first searches for the two shapes which contain the arrow ends, and then draws an arrow between these two shapes.

When one of the connected shape is transformed, i.e. moved, resized or rotated, the arrow connected with it will be updated, that is, its position, its length and its angle will be corrected, right after the transformation.

V. Selecting and operating on graphical elements

To select a single graphical element or several graphical elements one draws a circle around the element(s) he wants to select, using the right mouse button, only the elements whose bounds are entirely in the bound of the selection circle are selected. After selection the transparency of the elements' color is set to 0.5 to distinguish the selected ones from the unselected ones.

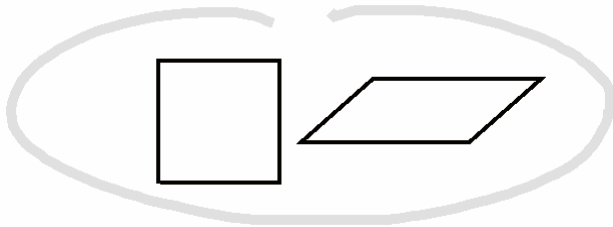


Figure 16. How to select graphical elements.

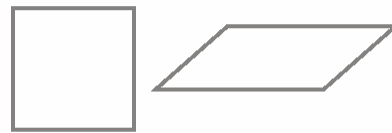


Figure 17. Screen output after selection.

After selection one can do operations on all these selected elements together. The operations can be selected from a pie-menu, which pops up when the user presses the right mouse button on an arbitrary place and holds it down for several milliseconds (default is 200 ms).

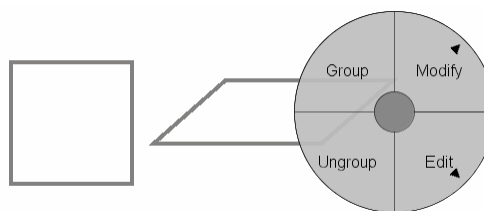


Figure 18. Pie-menu for operations.

Following operations are supported:

- Grouping selected graphical elements: If there are more than one graphical elements selected, this operation groups the selected graphical elements into a single one. The user clicks “Group” in the pie-menu shown in Figure 18 or draws the following gesture, using the right mouse button, to get this operation performed:



Figure 19. Gesture for group operation. The heavy dot indicates the starting point.

- Ungrouping selected graphical element: If there is only one graphical element selected and the element is the result of a previous group operation, then the subelements in this element become individual ones. The user clicks “Ungroup” in the pie-menu shown in Figure 18 or draw the following gesture, using the right mouse button, to get this operation performed.



Figure 20. Gesture for ungroup operation. The heavy dot indicates the starting point.

- Moving selected graphical element(s). To move the selected graphical element(s), the user clicks the right mouse button inside the selection area and holds it down, then drags it to the wanted place.
- Resizing and rotating selected graphical element(s). Whether a resize or a rotate operation is performed depends on where the user right-clicks the mouse button, s. Figure 21.

To start the resize operation, the user first right-clicks in the corresponding area, then within 500ms, the user draws a stroke using right mouse button inside this area, he should keep drawing until the wanted size is reached.

To start the rotate operation, the user first right-clicks in the corresponding area, then within 500ms, the user draws a stroke using right mouse button inside this area, he should keep drawing until the wanted angle is reached.

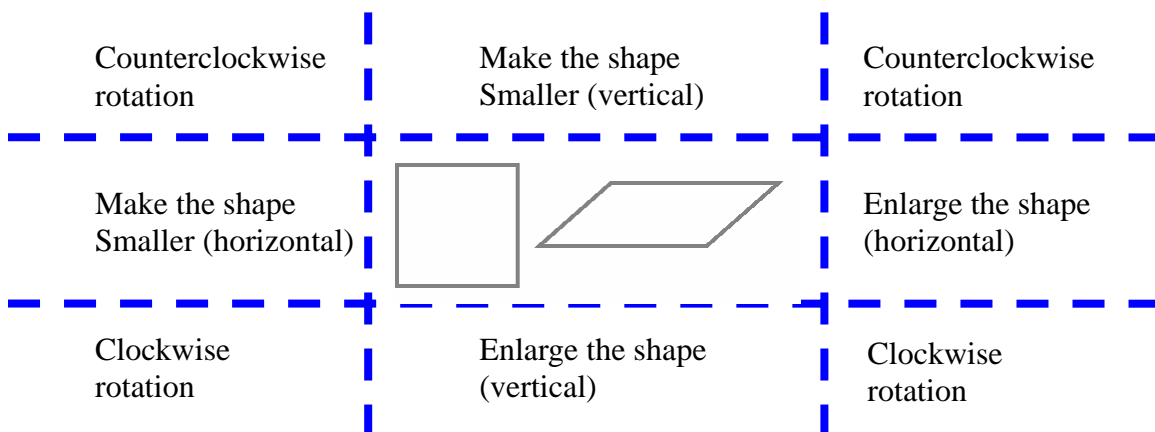


Figure 21. Different resize and rotate operations are assigned different areas to.

- Modifying selected graphical elements' appearance: make solid lines dashed, make dashed lines solid, choose another color for the elements, fill the elements with a certain color, change the line width. To perform the operation, one first selects the foreground color, background color, or line width, if necessary, in the corresponding controls, then clicks "Modify" in the pie-menu shown in Figure 18, right after that a child pie-menu pops up:

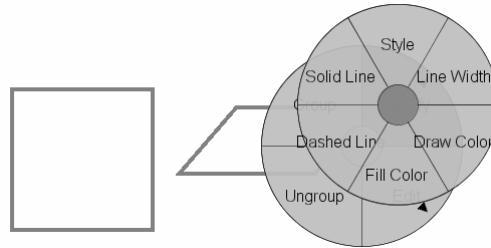


Figure 22. A child pie-menu for setting style of the selected graphical element(s).

- Cutting, copying, pasting, deleting selected graphical element(s). To perform these operations, the user clicks "Edit" in the pie-menu shown in Figure 18, right after that a child pie-menu pops up:

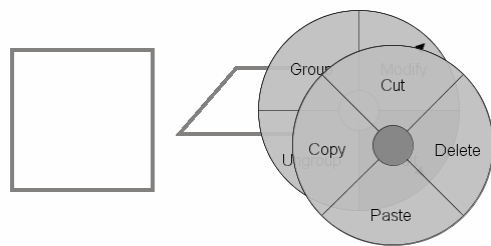


Figure 23. A child pie-menu for cutting, copying, pasting, deleting selected graphical element(s).

The operations *Cut*, *Copy*, *Paste* and *Delete* can also be called by drawing the following gestures on the screen:

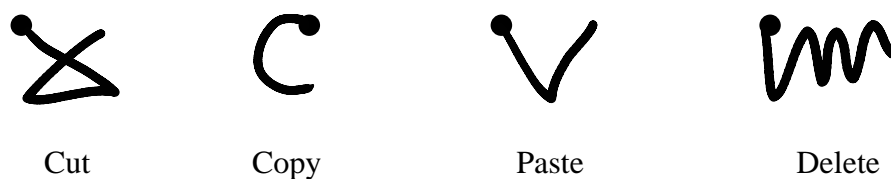


Figure 24. Gestures for cutting, copying, pasting, deleting selected graphical element(s).

To deselect the graphical element(s), the user draws again a circle around the graphical element(s) he wants to deselect, using the right mouse button.

Design Issues

I. Diagram items

All free strokes, geometrical shapes, text areas are instances of DiagramItem, it is an abstract class, and it inherits AbstractPropertyChangeSource. Properties of a diagram item are HAS_BEEN_REMOVED, LOCATION_PROPERTY, SIZE_PROPERTY and STYLE_PROPERTY. Each diagram item has a unique identification number and a GOBWrapper object for its compatibility with SATIN's GraphicalObject.

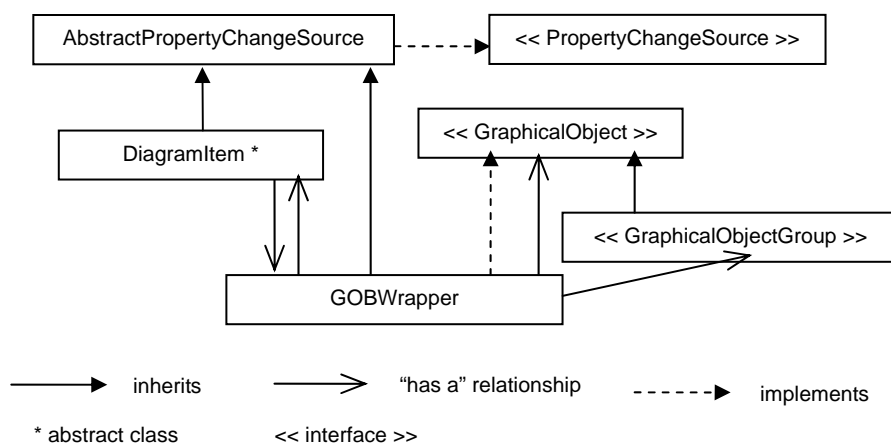


Figure 25. DiagramItem class.

Several diagram items can be grouped into a DiagramItemGroup object. DiagramItemGroup inherits DiagramItem, and it maintains a list of DiagramItem instances which belong to this group, as a consequence, it implements PropertyChangeListener and listens to the HAS_BEEN_REMOVED property change event of each diagram item contained in it, when a corresponding PropertyChangeEvent is fired, the event source will be deleted from the group. Each instance of DiagramItemGroup has a GOBGroupWrapper object for its compatibility with SATIN's GraphicalObjectGroup which inherits GraphicalObject. GOBGroupWrapper inherits GOBWrapper.

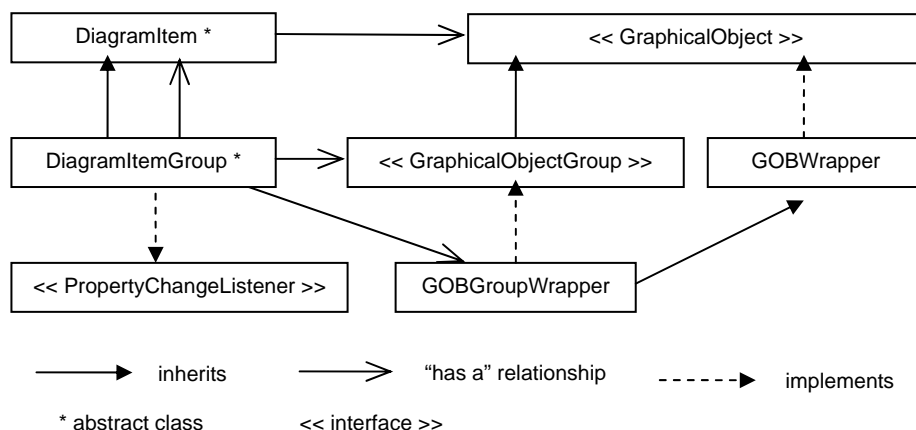


Figure 26. DiagramItemGroup class.

Following are the classes for geometrical shapes which are created in basic shape mode, complex shape mode and text mode, these classes are in the package `brainchild.editor.shared.items`:

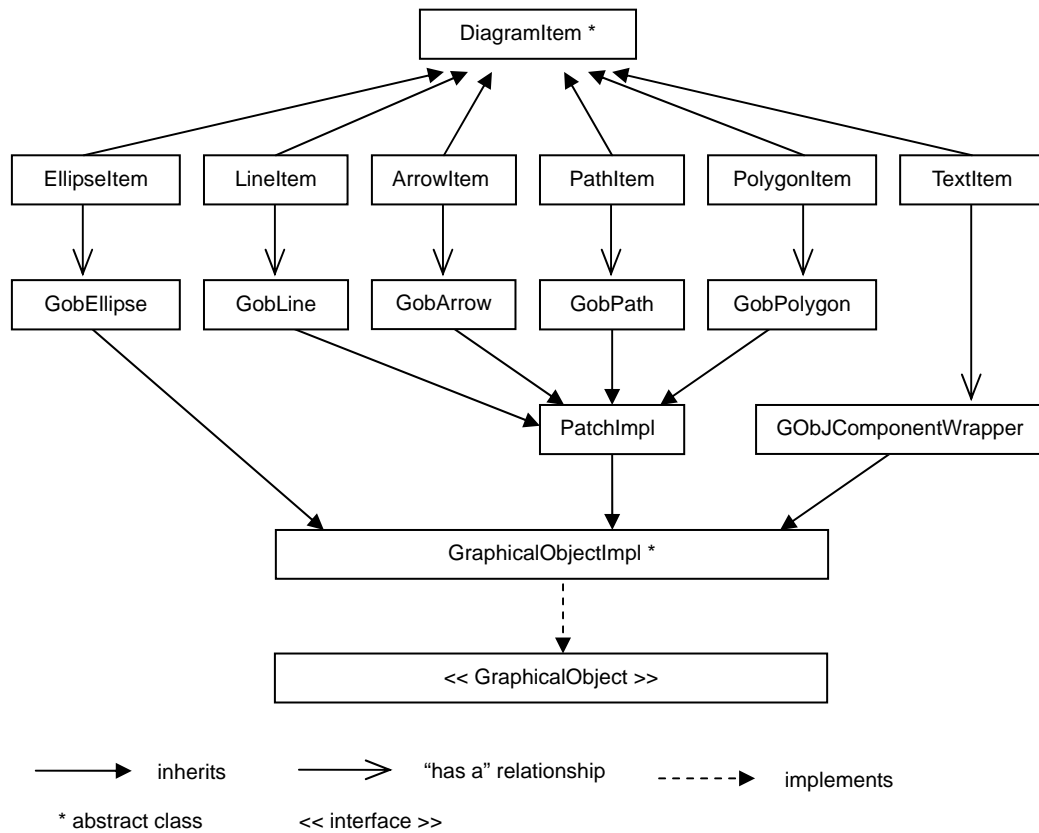


Figure 27. Classes for geometrical shapes.

When a shape is drawn, an object of the corresponding class `XxxItem` is created, the default `tRender()` method which is inherited from `GraphicalObjectImpl` and is responsible for rendering the shape on the screen is implemented in the `PatchImpl` resp. `GobXxx` class. Concretely,

- Ellipses, including circles, are objects of `EllipseItem`.
- Arrows are objects of `ArrowItem`.
- Lines are objects of `LineItem`.
- Rectangles, squares, triangles, parallelograms and rhombuses are objects of `PolygonItem`.
- Paths which are constructed from straight lines and are not rectangles, squares, triangles, parallelograms or rhombuses are objects of `PathItem`, the user can draw paths only in complex shape mode.
- Text areas created in text mode are objects of `TextItem`.

The objects of the geometrical shapes described above are all created using methods implemented in the `ShapeFactory` class.

Different instances of DiagramItem can be grouped together using group operation. A group is an instance of ItemsGroup which is a subclass of the abstract class DiagramItemGroup:

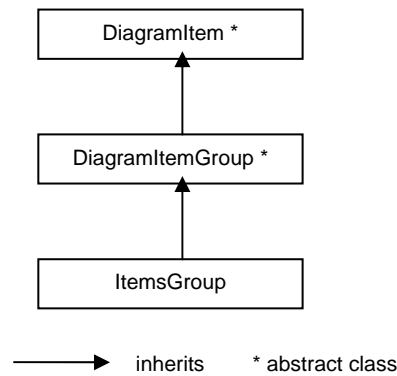


Figure 28. Classes for groups of DiagramItem

II. Editor modes and their associated interpreters

In a specific editor mode one can only draw a predefined set of DiagramItem, and process a predefined set of gestures. In the current project, following editor modes are implemented: DiagramEditorMode, DiagramBasicEditorMode, DiagramComplexEditorMode and TextEditorMode, where DiagramBasicEditorMode and DiagramComplexEditorMode are submodes of DiagramEditorMode. These classes are in the package brainchild.editor.modes:

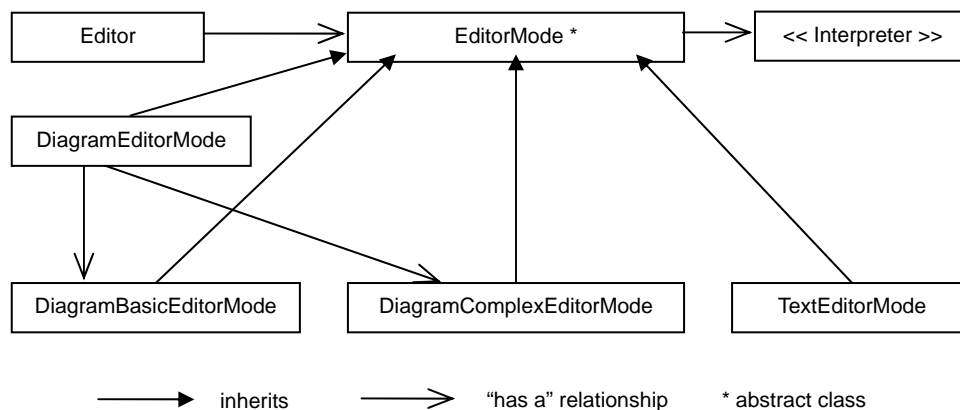


Figure 29. Classes for editor modes.

Each editor mode is associated with its own gesture and ink interpreters. When the user enters a new editor mode, the interpreters associated with this editor mode are initialized. So each time when the user draws a stroke resp. a gesture, a list of ink resp. gesture interpreters calls their handleNewStroke(), handleUpdateStroke() and handleSingleStroke() methods inherited from InterpreterImpl in the order in which they are added to the list.

The classes for ink and gesture interpreters are in the package brainchild.editor.shared.interpreters.

`BasicShapesInkInterpreter` and `BasicShapesGestureInterpreter` are associated with `DiagramBasicEditorMode`. Both are based on `RubineInterpreter` which is used to classify strokes.

`BasicShapesInkInterpreter` recognizes different types of geometrical shapes like lines, rectangles, squares, ellipses, circles, triangles, parallelograms, rhombuses, solid lines with a solid arrowhead, solid lines with an open arrowhead. If the probabilities of all these classifications are under the threshold 0.8, then the interpreter assumes that the user wanted to draw a line.

`BasicShapesGestureInterpreter` recognizes gestures which are drawn to indicate arrow types: solid line with a left resp. right solid arrowhead, solid line with a left resp. right open arrowhead, solid line with double solid arrowheads, solid line with double open arrowheads. This interpreter also plays a role in context-aware drawing, see below.

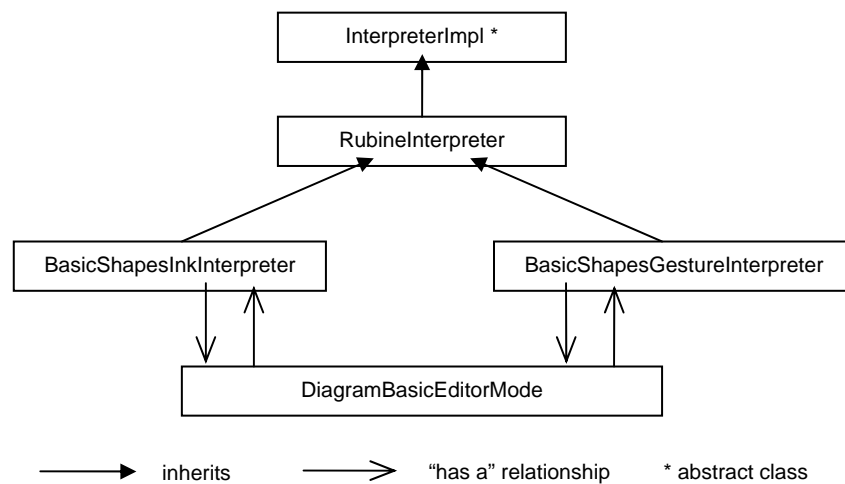


Figure 30. Interpreter classes in basic shape mode.

`ComplexShapesInkInterpreter` and `ComplexShapesGestureInterpreter` are associated with `DiagramComplexEditorMode`. Both inherit the abstract class `InterpreterImpl`.

`ComplexShapesGestureInterpreter` keeps track of whether the user has entered drawing phase or not, the program is in the drawing phase when a “drawing” text can be seen on the screen.

`ComplexShapesInkInterpreter` passes free strokes to `DiagramComplexEditorMode` when the program is in the drawing phase, and `DiagramComplexEditorMode` stores all these strokes drawn in a certain drawing phase, straighten these strokes to lines and output the lines on the screen. When the user has finished drawing the complex shape, `DiagramComplexEditorMode` processes these lines, trying to classify the shape: if the shape is open, then an instance of `PathItem` is created; if the shape is closed and can be classified as square, rectangle, rhombus or parallelogram, then an instance of `PolygonItem` is created, otherwise an instance of `PathItem` is created.

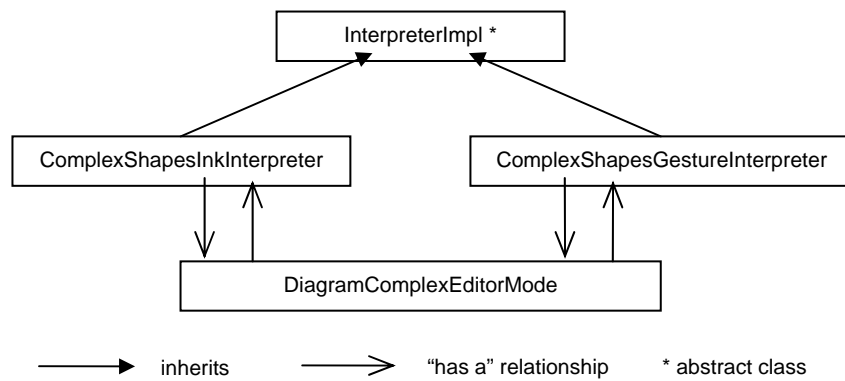


Figure 31. Interpreter classes in complex shape mode.

TextBoxInterpreter is associated with TextEditorMode. It inherits the abstract class InterpreterImpl. TextBoxInterpreter is responsible for creating instances of TextItem and passing them to TextEditorMode. TextEditorMode stores all the instances of TextItem present and keeps track of the current TextItem, its appendCharacter() method is called by SmallLettersInterpreter, BigLettersInterpreter, NumbersInterpreter and PunctuationsInterpreter which are all subclasses of RubineInterpreter and are associated with the four circular controls TextSmallLettersControl, TextBigLettersControl, TextNumbersControl and TextPunctuationsControl. The last four interpreters are used to recognize letters, numbers and punctuations.

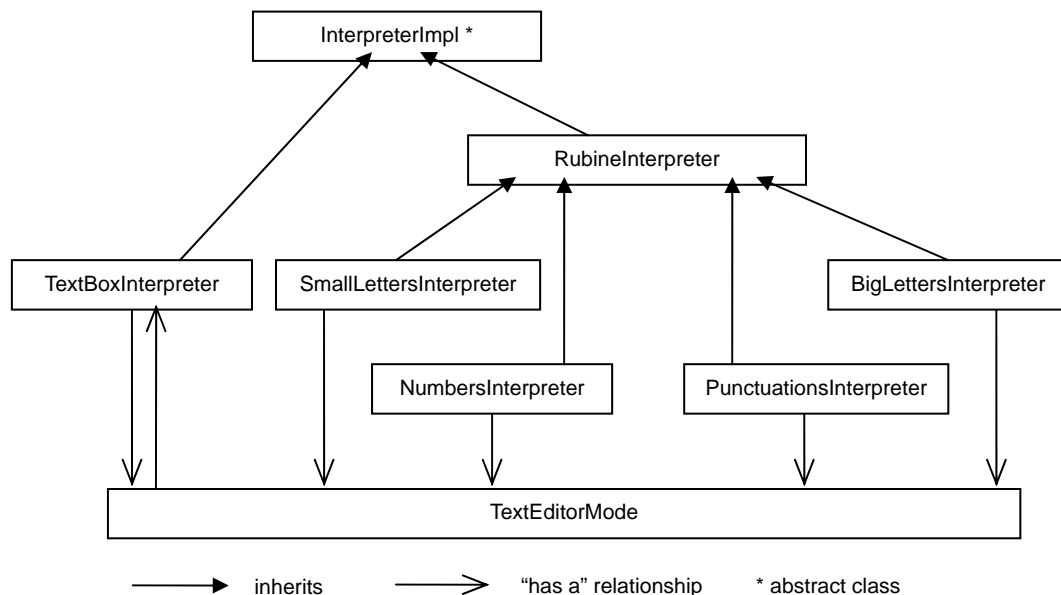


Figure 32. Interpreter classes in text mode.

III. Other interpreters

There are other four gesture interpreters which are directly associated with the instances of `DiagramSatinSheet` and are therefore independent of the editor modes. They are `Selecti onGestureInterpreter`, `MoveSel ectedInterpreter`, `Resi zeRotateSel ectedInterpreter` and `Edi tGestureInterpreter`. The first three are subclasses of `InterpreterImpl`, and the fourth one is a subclass of `Rubi neInterpreter`.

`Selecti onGestureInterpreter` recognizes select and deselect operations, keeps track of selected instances of `DiagramI tem`, it also offers methods for grouping, ungrouping, styling, copying, cutting, pasting and deleting selected elements which are called by `Edi tGestureInterpreter` or the pie-menu of an instance of `DiagramSati nSheet`.

`MoveSel ectedInterpreter` recognizes move operations by checking whether the user right-clicks inside the selection area and draws some stroke. If yes, then the selected elements are moved to the wanted position.

`Resi zeRotateSel ectedInterpreter` recognizes resize and rotate operations. If a resize resp. rotate operation is recognized, then the selected elements are resized or rotated until the user stops drawing the gesture stroke.

`Edi tGestureInterpreter` recognizes commands for grouping, ungrouping, copying, cutting, pasting and deleting selected elements. If one of these operations can be recognized, then it calls the corresponding method in `Selecti onGestureInterpreter`.

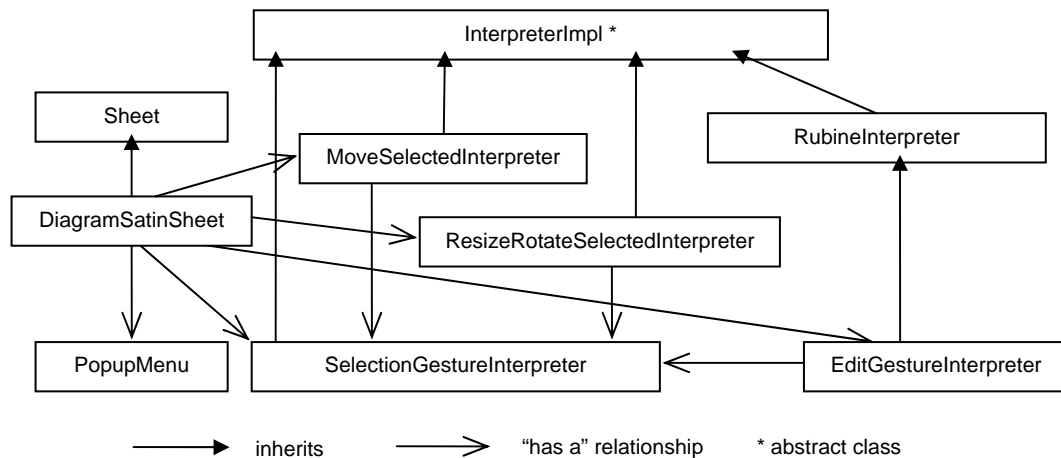


Figure 33. Interpreter classes associated with `DiagramSati nSheet`.

IV. Context-aware drawing

For the subclasses of `DiagramI tem` which are connectable by arrows they should implement the interface `ArrowConnectabl e`, it is in the package `brai nchi l d. edi tor. shared. i tems`. `Pol ygonI tem` implements `ArrowConnetabl e`. `ArrowConnectabl e` has two methods:

```
publ i c Rectangl e2D getBound();
```

```
public Point2D getArrowPoint(ArrowConnectable dest);
```

When `BasicShapesGestureInterpreter` recognizes an arrow, it first searches for two instances of `ArrowConnectable` each of which contains one arrowhead. If they can be found, then the interpreter gets an arrow with the correct position, length, angle created and passes these two instances of `ArrowConnectable` to the arrow object; If they cannot be found, then it gets an arrow with the original position, length, angle created.

`ArrowConnectable` also extends the interface `PropertyChangeSource`, and during initialization of an arrow, the two connected shapes are registered in the instance of `ArrowItem` and after that the arrow is added as `PropertyChangeListener` to the list of `PropertyChangeListeners` associated with each `ArrowConnectable`:

```
connection.addPropertyChangeListener(DiagramItem.HAS_BEEN_REMOVED, removedListener);

connection.addPropertyChangeListener(DiagramItem.LOCATION_PROPERTY,
locationListener);

connection.addPropertyChangeListener(DiagramItem.SIZE_PROPERTY, locationListener);
```

In this way, the arrow is updated as soon as one of the connected shapes changes its location, its size or its direction, and the arrow is deleted after one of the connected shapes is deleted.

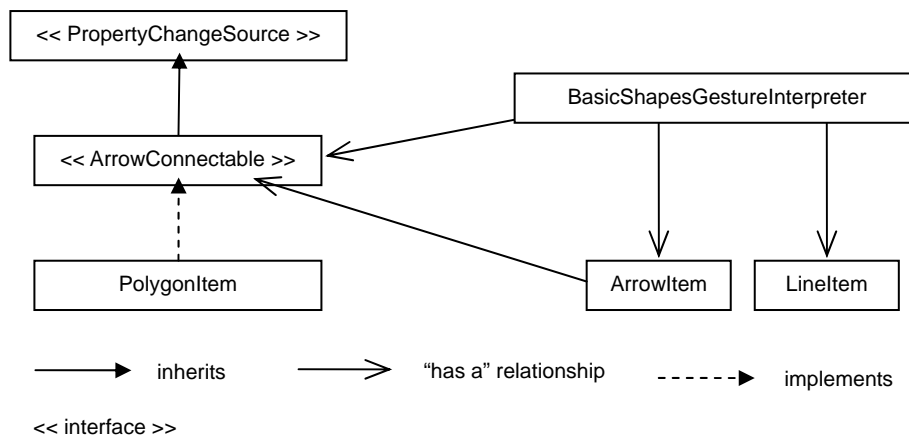


Figure 34. Classes involved in context-aware drawing.

V. Controls for selecting modes

The classes `Mode` and `EditorMode` are in the package `brainchild.editor`.

To instantiate an instance of `Mode`, one must specify a string for tooltip, and three image files which are presented for the cases that the mode is not the current one, the button for the mode is pressed, and the mode is the current one.

The abstract class `EditorMode` inherits `Mode`, its subclasses represent different editor modes. An editor mode can have several submodes, like `DiagramEditorMode` which has `DiagramBasicEditorMode` and `DiagramComplexEditorMode` as its submodes. So `EditorMode` is a client of itself.

To select an editor mode, the user uses `EditorSelectionControl` and `DiagramTypeSelectionControl`, if needed. They are implemented in the package `brainchild.ui.controls`.

The abstract class `SelectionControl` is for selecting modes, it is circular, and it inherits `AbstractCircularControl`. `SelectionControl` is responsible for the layout, and its getter and setter methods can be called to get or set the current mode.

Using `EditorSelectionControl` one can switch between freehand drawing mode, non-freehand drawing mode and text mode. All these modes are instances of `EditorMode`. `EditorSelectionControl` inherits `SelectionControl`.

When non-freehand drawing mode is activated, an instance of `DiagramTypeSelectionControl` is created. Using this control one can switch between basic shape mode, complex shape mode and flow graph mode (the last one is not implemented in the current project). All these modes are instances of `EditorMode`. `DiagramTypeSelectionControl` inherits `SelectionControl`.

The classes with graphical rendering methods for the controls are implemented in the package `brainchild.ui.controls.plaf`. They are `AbstractCircularControlUI` and `SelectionControlUI`, the latter inherits the former.

All the controls described above are put into the `DashBoardPanel` in the package `brainchild.ui.panels`.

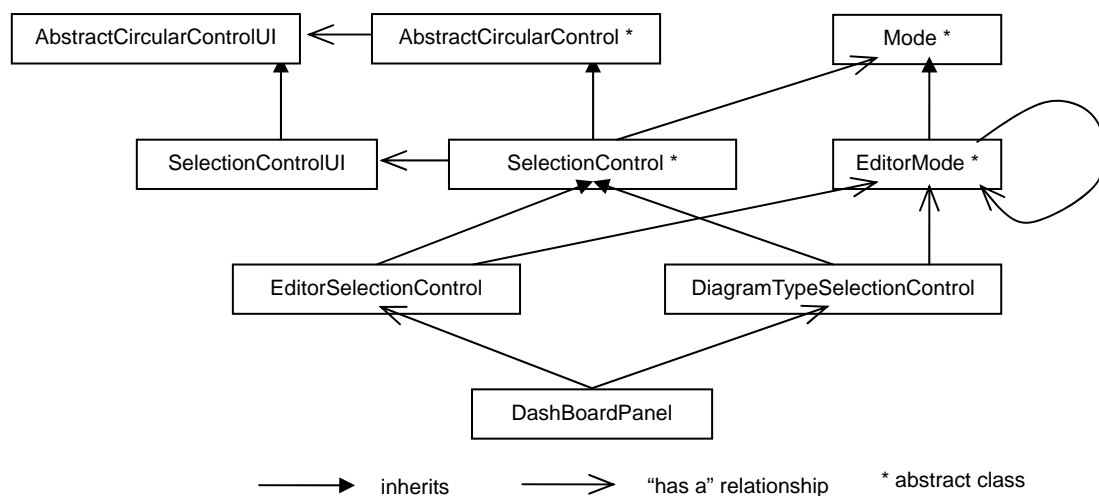


Figure 35. Classes for selection controls.

VI. Controls for text mode

When text mode is activated, four circular controls pop up on the screen, they are `TextSmallLettersControl`, `TextBigLettersControl`, `TextNumbersControl` and `TextPunctuationsControl`, all of them are subclasses of the abstract class `TextControl`. Each instance of `TextControl` has its own `Sheet` object with which an interpreter for recognizing letters, numbers and punctuations is associated. These controls are circular, they inherit `AbstractCircularControl`.

The classes with graphical rendering methods for the controls are implemented in the package `brainchid.ui.controls.plaf`. They are `AbstractCircularControlUI` and `TextControlUI`, the latter inherits the former.

All the controls described above are put into the `DashboardPanel` in the package `brainchid.ui.panels`.

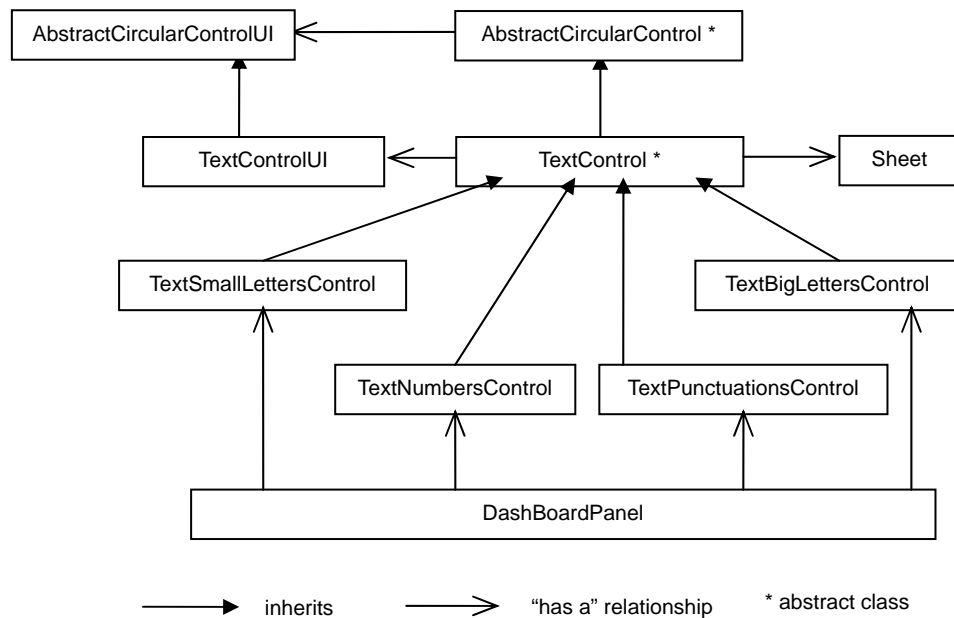


Figure 36. Classes for text mode controls.

Results

The work so far has reached most of the intended results; however, there also exist several bugs. Following problems probably originate from SATIN's implementation:

- Problems with `DiagramItemGroup` resp. `GOBGroup` resp. `GraphicalObjectGroup`:
 - Styling a `DiagramItemGroup` does not work. This problem results that when a group is selected, the transparency of the elements' color cannot be set, so the user cannot distinguish a selected group from an unselected group optically.
 - When resizing or rotating a `DiagramItemGroup`, the `DiagramItems` in the group are not updated, so after ungrouping them, these individual `DiagramItems` return to the original size or the original direction.
 - In case of a group of `DiagramItemGroups` the method `getCollectionBound2D()` returns `(0, 0)` as the group's upper left point, although the entire group stays rather in the middle of the screen. This problem affects selection and deselection.
 - The `clone()` method of `GraphicalObjectGroupImpl` is probably not correctly implemented, a `java.util.ConcurrentModificationException` is thrown when pasting a `GraphicalObjectGroup`.

- A text area in text mode is an instance of `GobJComponentWrapper` implemented in SATIN which is a wrapper for java swing's `JComponents`. The problem is that the author must first add some letters, i.e. '-'s, to get an `JTextArea` object extend to the intended width, and then pass it to the wrapper, otherwise the width of the `JTextArea` is always zero, and no characters can be added to it, in other words, the bound of a `JTextArea` does not change when the user adds characters to it, and it always only refers to the first line, as a consequence, a text area can only be activated when the user left-clicks on the first line, and when it becomes the current text area, only the first line is highlighted with a yellow background.
- The user first selects an element, then right-clicks in the selection area to get the pie-menu, then closes the pie-menu, then the user right-clicks outside the selection area, then one can observe that the selected element moves. The cause is that with the second right-click, `MoveSelectedInterpreter.handleNewStroke()` is not called before `MoveSelectedInterpreter.handleUpdateStroke()` is called, so `isOverSelected = true` is the value set during the first right-click, and this boolean flag with the value `true` leads to the update of the element's position.

Following problems have probably to do with the current implementation:

- **GobLine:** In the current implementation, `GobLine` extends `PatchImpl` provided by SATIN, the problem with this approach is that in basic and complex shape modes, horizontal and vertical lines are not output on the screen. To solve this problem, one can let `GobLine` extend `GraphicalObjectImpl`, but then it comes to the following problem:
- **GobEllipse, GobLine:** In the current implementation, `GobEllipse` extends `GraphicalObjectImpl` provided by SATIN, the problem with this approach is that after `GraphicalObjectImpl.defaultRender()` is called, the actual internal coordinate of an `GobEllipse` resp. `GobLine` object is not the same as the one observed on the screen.

Remarks

The author had underestimated the time and effort she needed to complete the project. So redo and undo operations for the diagram editor had to be canceled due to lack of time.

The main reason for the underestimation was that the framework SATIN was a totally new field for the author, and it took her some time to get familiar with, considering that most classes, including their fields and methods, were not well documented and there also exist several bugs with SATIN's implementation. Another time-consuming work was to calculate coordinates of intersection points, connection points for arrows, and something like that.

References

[1] SATIN - A Toolkit for Informal Ink-based Applications; Online at:
<http://exuma.cs.berkeley.edu/projects/satin/>, consulted in July – October 2006.

[2] The JavaTM Tutorials, Trail: 2D Graphics; Online at:
<http://java.sun.com/docs/books/tutorial/2d/index.html>, consulted in July – October 2006.