Scripting Diagrams with *EyeSee*

Matthias Junker, Markus Hofstetter
Software Composition Group
University of Bern, Switzerland

June 1, 2007

Abstract

Presenting numbers in the right way is crucial for understanding their meaning. However, many existing diagram drawing tools do not make understanding the numbers as easy as it could be. They often insert too many visual distractions or require a fixed input format. *EyeSee* is a model-independent diagram drawing engine that allows for programmatic specification of the presentation, while offering default values that produce uncluttered diagrams. As a validation, we demonstrate the simplicity to create well-known diagrams.

Contents

1 Introduction **3**
1.1 Document structure 4

2 *EyeSee* by Example **5**

3 *EyeSee* Internals **9**
3.1 Decorators. 11

4 *EyeSee* Validation **13**
4.1 Horizontal Bar Diagram 13
4.2 Vertical Bar Diagram 13
4.3 Composite Diagrams 13
4.4 Deviation Diagram 15
4.5 Scatterplot 16
4.6 Line Diagram 17
4.7 History Diagram 19
4.8 Stack Diagram 19
4.9 Range Diagram 20

5 Field study and future work **22**
5.1 MooseDen 22
5.2 Interactivity 23

6 Conclusion **24**

7 Appendix **25**
7.1 Quick Reference 25
7.2 Availability 27

1 Introduction

Presenting numbers is one of the most important practices in business and science. Because much of our perception is dedicated to vision, it is often more natural to extract the meaning of numbers from a well drawn picture than to reason about the numbers themselves [3]. Diagrams and tables are powerful tools to help people understand the patterns, relationships and trends in numbers. They exploit human's ability to recognize patterns and the capability of extracting a lot of information in a short period of time from familiar visualizations.

However, reading and understanding diagrams is not always as easy as it could be. If the important data is distracted by too many unnecessary visual elements, it can be hard to get the numbers. Edward Tufte and Stephen Few wrote books about this topic, giving advice on creating accurate diagrams [1, 2]. They suggest general rules which should be applied in every kind of diagram and also provide examples for the most common kind of diagrams. The most important suggested rule is: *Reduce Chart Junk and Maximize Data Ink* [2].

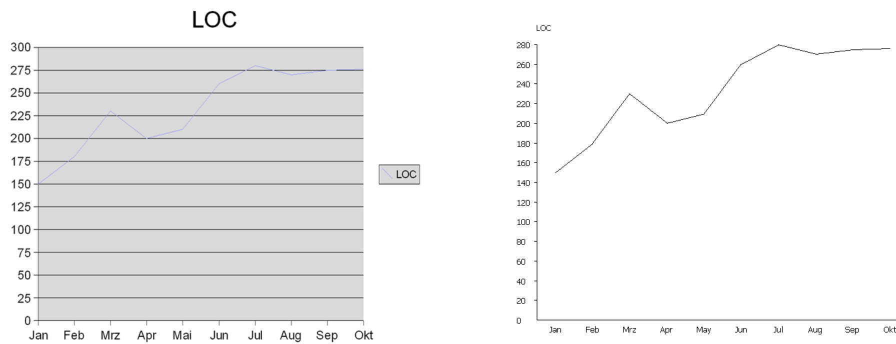
This means that a good diagram should eliminate chart chunk like dominant background colors or fancy 3D effects that do not convey actual information.

With *EyeSee*, we want to introduce a diagram drawing engine which provides a solution for the following aspects:

Flexibility. *EyeSee* does not require the data to be passed in a fixed format. The user can specify programmatically how to extract the data, which gives him the freedom to accommodate any data model he is using.

Scriptable. Maybe the user does not know from the beginning which type of diagram is the most suitable for his data or he is not satisfied with the look of the automatically generated visualization. This is why he should be allowed to change the parameters of the diagrams with little effort. For this purpose we provide scripting methods, with which you can change most of the properties of a diagram (e.g., the color of the elements, the type of axis, the size etc). Our goal is to generate the diagrams in a way, so the user does not *have* to script anything but still *can*, if he chooses to.

Defaults. The line diagram on the left side was generated by the diagram wizard of Excel while the one on the right was created by *EyeSee*. With *EyeSee* we implemented a diagram engine that aims to produce uncluttered diagrams by default, so that the user does not have to change the default values to get a result that focuses on the data.



1.1 Document structure

In Chapter 2 we give an example of how the user can script a diagram with *EyeSee*.

In Chapter 3 we present the internals of *EyeSee* and talk about some of the issues we had to solve.

In Chapter 4 we validate *EyeSee* by showing how to create various established diagrams.

In Chapter 5 we give an outlook on the directions we plan to go with *EyeSee*.

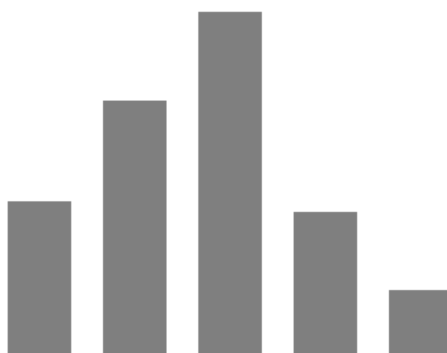
In Chapter 6 we recapitulate the problem and explain how we tried to solve it with *EyeSee*.

2 *EyeSee* by Example

In this section we show the basic facilities of *EyeSee* using hands-on examples. For building a model, we require data, which we extract from a model. This is in most cases a collection of objects, which contain the data. Our model in this example is a collection called *contributors* containing the contributors to a certain project, as obtained from a versioning system. A contributor has the attributes *name*, *number of commit*s, *lines of code*, *team* and *versions*.

Basic Diagram. In our first example we want to create a vertical bar diagram which shows us the commit activity of all developers. By providing a block or a symbol which represents a selector, we tell *EyeSee* how to extract the data from a contributor which will be used for the height of the corresponding bar.

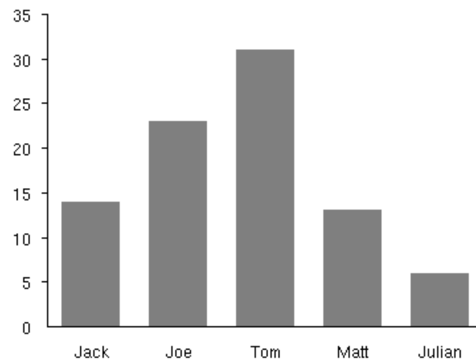
```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
    y: #numberOfCommits;
    models: contributors.
diag open
```



Axis and Labels. As we see, it is a vertical bar diagram, but it is still useless because we cannot see which bar is from which contributor, and we cannot get the actual values from the diagram. So we want to add some axis, and label the bars with the name of the contributor. We can do this with two additional lines:

```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
    y: #numberOfCommits;
    regularAxis;
    identifier: #name;
    models: contributors.
diag open
```

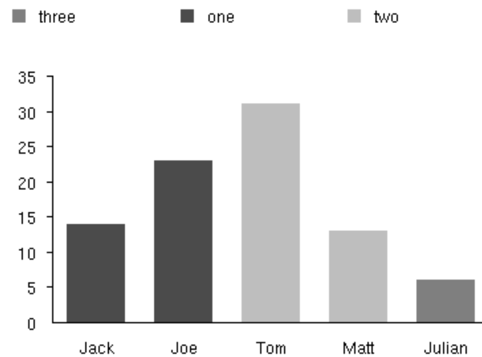
`regularAxis` adds an axis with ticks and labels. "Regular" means that the distance between two ticks is constant. Instead, we could also use `valueAxis`, which draws ticks only if there is a corresponding value in the diagram. This



can make it easier to tell at a glance where the majority of the values are located in a diagram. To put a label below every bar, we can use `identifier:`. In our example, we use the name of the contributor to identify the bars.

Color. If we want to encode another attribute in our diagram, we have this possibility by coloring or shading the bars differently. In our example we want to use different colors to show, who belongs to which team. By default, the values for the color get encoded with shades of gray. By sending the message codelightColors or `strongColors`, we can use a set of ten colors which are distinct enough so they can be easily separated.

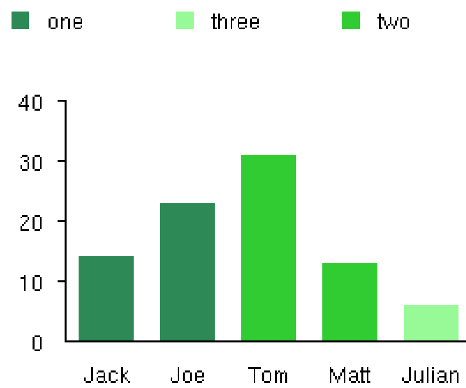
```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
    y: #numberOfCommits;
    identifier: #name;
    color: #team;
    regularAxis;
    models: contributors.
diag open
```



If we want to provide our own colors for the encoding, we can use `useColors:`.

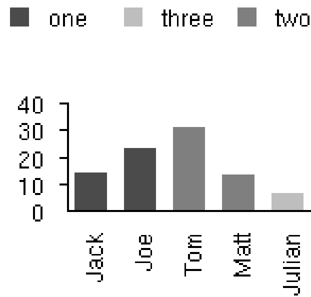
```
diag := DiagramRenderer new.
```

```
(diag verticalBarDiagram)
    identifier: #name;
    y: #numberOfCommits;
    color: #team;
    useColors: #( #seaGreen #limeGreen #paleGreen );
    regularAxis;
    models: self model.
diag open
```



Size. To change the default size of diagrams, we can specify this width and height. If there is not enough space for the identifiers to be displayed horizontally, they automatically get rotated.

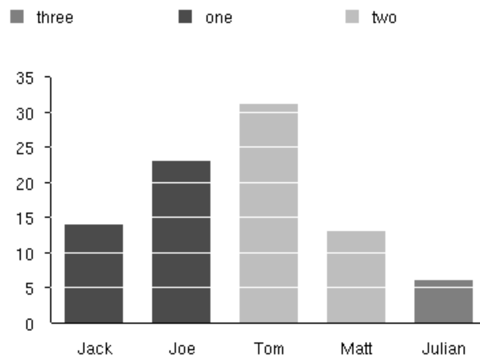
```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
    identifier: #name;
    y: #numberOfCommits;
    color: #team;
    regularAxis;
    width: 200;
    height: 200;
    models: self model.
diag open
```



Of course we can also configure to have them rotated manually by using the `setRotatedLabels` message. Optionally you can also change the padding of the diagram with `xPadding:` and `yPadding:`.

Grid lines. It is not always easy to estimate the values displayed in a diagram, especially when the axis with the values are far away from the actual data elements (e.g. bars). Putting a grid over the actual content is the usual approach to that problem. But since we want to minimize non-data-ink, we adopted the approach from Tufte [2]:

```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
    identifier: #name;
    y: #numberOfCommits;
    color: #team;
    regularAxis;
    gridLine;
    models: self model.
diag open
```



3 *EyeSee* Internals

Wrapper, Decorator, Chain of Responsibility, UML,

Diagram. The *AbstractDiagram* class is the root of our diagram hierarchy. It contains the model of the data and the logic that creates the painting of the diagram. The model can be set using the `models:` message. We assume this to be a collection of the model objects that the user is working with. To let the diagram know how to extract the data from the model object we use the `y:` (and in the case of a scatter plot which displays two properties also the `x:`) message. The parameter is a block or a symbol representing a selector that, when sent to one of the model objects will let it answer the relevant value. It is mandatory to set those two attributes (or three in the case of a scatterplot) if the user wants to create a meaningful diagram. For the other attributes of the diagrams like width, line color or padding we provide reasonable default values. Once this is done, we can call the `setup` method to properly initialize the axis and the elements of a diagram. The elements of a diagram encapsulate the graphical representation of the values with the corresponding model object. For example an element in a bar diagram consists of a rectangleShape (the bar), the color we want to draw this bar in and the model object that belongs to this bar. If an element receives the `displayOn:` message, it simply forwards it to its shape. The possibility to return the underlying model object allows for some interactivity.

The *DiagramRenderer* class provides scripting methods to create our diagrams. When it receives the `open` message it prepares the diagram for drawing by calling its `setup` method. Then it opens a *VisualizationUI* containing the diagram. *VisualizationUI* is a class that belongs to the CodeFoo package. It enhances our visualization with a context menu that gives access to some of its features such as exporting our diagrams to the .png format or zooming. It also comes with a status bar that we use to show the name of the model object that belongs to the figure that the mouse is hovering over. Since *EyeSee* is written in Smalltalk it is also possible to inspect the model objects behind a figure at runtime by choosing inspect in the context menu.

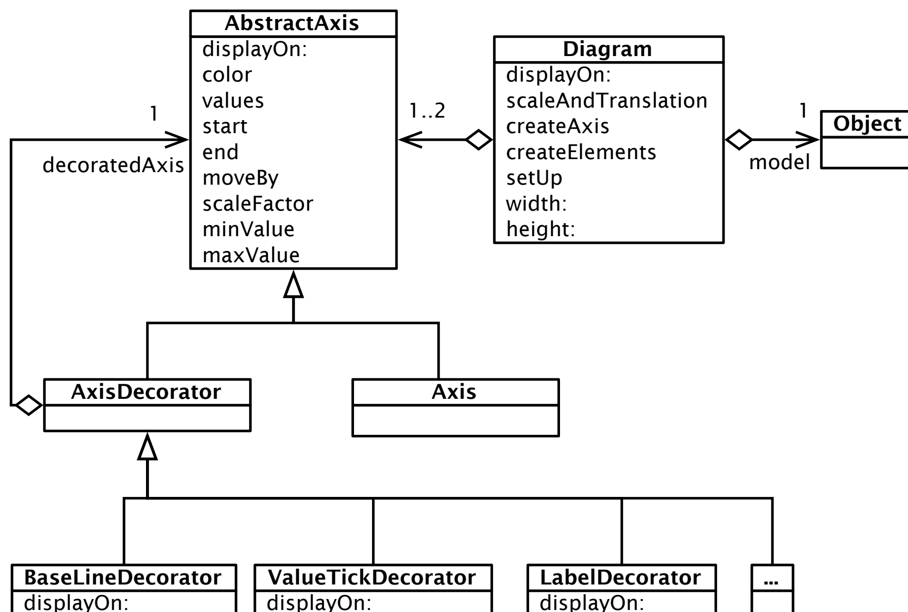
Scaling. There are three groups of instance variables that relate to the size of a diagram; width/height, borders and padding. The user can adjust the overall size by setting the width and height attributes. They determine the maximum space that the diagram may occupy. *EyeSee* scales the diagram to fit in this space. If the user wants a padding around his diagram he can specify an x- and y padding that will create whitespace around the diagram.
The borders are calculated automatically. The border is the space *EyeSee* is using for things like the labels of the axis or identifiers (labels of bars). If there are long identifiers *EyeSee* will assign more space to the border. For example if the identifiers are month names fully typed out (instead of abbreviated) *EyeSee* will widen the border a bit and the diagram itself will be drawn on a slightly smaller space to ensure that the specified width and height will not be exceeded.

The size of padding, border and the diagram itself will add up to the extent

that is specified via width and height. This is useful if the user wants to create a diagram of a certain size. All he has to do is set the width and height of the diagram and specify the padding (if he wants padding).

Translation. An issue we encountered while creating *EyeSee* was that the coordinate System of Smalltalk has its origin in the upper left corner of the screen and its y axis grows downwards. To model the graphical representation of our diagrams more naturally, we decided to change the coordinate system to have a y axis that grows upwards. Also the origin is usually translated to the origin of the diagram. To achieve this we implemented a wrapper for the class GraphicsContext that is responsible for drawing in VisualWorks. Whenever the `displayOn:` method of a diagram is called we wrap the graphicsContext that is passed in our *DiagramGraphicsContextWrapper* before we draw anything. This *DiagramGraphicsContextWrapper* transforms every shape or point that is about to be drawn by the scale point and translation that it stores. Then it forwards the translated shape to the GraphicsContext for drawing. Usually this means that the y axis is inverted and that the origin is translated down by the height of the diagram. However, the *StackDiagram* for example is created from top to bottom and does not invert the y axis.

It is also possible to translate the axis by a different amount than the content of a diagram. A typical use case for this is a *VerticalBarDiagram* that contains negative values. In this case the x axis must be moved up to the zero on the y axis. To allow for these multiple layers every diagram has a *layerHolder* that stores the multiple wrappers. For every layer there is one wrapper (e.g., one wrapper for the content and one for each axis). When drawing the diagram, we change the active wrapper every time we draw a part of the diagram that has its own layer.



Axis. The *Axis* class contains the collection of values that are relevant to this axis (e.g. the y-axis of a horizontal bar diagram knows about the height of the bars) and some other properties like the color. But an axis has no knowledge of the other parts of the diagram (diagram and axis).

To allow for the axis of the diagrams to be displayed in various manners according to the needs of the user, we implemented a decorator pattern. Every axis can be wrapped in several decorators. The decorators and the axis are subclasses of *AbstractAxis* and every decorator knows the axis/decorator it decorates. If a decorator receives the `displayOn:` message it forwards it to the axis/decorator and then displays itself. This leads to a chain of responsibility where every decorator only draws the parts of the axis which it is responsible for. We provide some basic decorators (*e.g., RegularTickDecorator, ValueTickDecorator, LabelDecorator, BaseLineDecorator etc.*) that can be composed by the user.

The benefit of this pattern is that the displaying behavior of the axis can be changed depending on the used decorators. For example we could show Ticks on the Axis in regular intervals or by changing the decorator we could show them only if there is a corresponding value in the diagram. This also allows for extensibility because if there is a new idea on how the axis should be drawn the only thing that needs to be done is the implementation of a decorator that responds to the `displayOn:` message in the desired way.

Since it can get tedious to set up this "chain" of decorators all the time to create the standard looking axes that are used very often, we provide scripting methods in the diagrams that will create them automatically.

3.1 Decorators.

This list introduces briefly all of the decorators we implemented and what they will display.

- *BaseLineDecorator*: this is the most basic decorator. It simply draws a straight line for the axis.
- *RangeBaseLineDecorator*: this draws a straight line from the minimum Value of the axis to its maximum. This is useful to show the range of the values.
- *RangeLabelDecorator*: displays the labels for the values at the maximum and the minimum of an axis.
- *FewBoxPlotBaseLineDecorator*: displays the median, upper and lower quartile and the range of the data on the axis as proposed by Few [1]. Those four important points on the axis are marked by slightly shifting the line.
- *BoxPlotBaseLineDecorator*: does the same as the FewBoxPlotBaseLineDecorator but draws the box instead of only shifting the lines.
- *BoxPlotLabelDecorator*: draws four labels with the values at the points that are emphasized by the box plot (i.e., max, min, median, upper and

lower quartile).

- *IdentifierDecorator*: responsible for drawing the identifiers of a diagram.
- *ValueTickDecorator*: draws a tick on the axis whenever there is a corresponding value in the diagram. For example it will draw a tick at the height of every bar when it is added to a *VerticalBarDiagram*.
- *ValueLabelDecorator*: draws a label for every value in the diagram just like the *ValueTickDecorator* does for the ticks. If there are many values close together, the labels would overlap when drawing every one of them. To solve this issue the decorator checks if the labels would overlap before drawing them. If this situation occurs on a x axis it rotates the labels by 90. If it occurs on a y axis it leaves out some of the labels to gain space.
- *RegularTickDecorator*: draws ticks on the axis that always have an equal distance from each other. It divides the axis into intervals of the same size.
- *RegularLabelDecorator*: most of the time this decorator is used in conjunction with the RegularTickDecorator. It is smart enough to choose an interval size that is a multiple of 5 and that gives the labels enough space so that they do not overlap.
- *RegularGridLineDecorator*: this decorator draws a white line over the whole width of a diagram wherever there is a regular tick on the y axis. This is useful in bar diagrams because it makes it easier to estimate the height of a bar.
- *LabelDecorator*: this draws the label of an axis (e.g., LOC ,NOM)
- *StackedValueDecorator*: this decorator is only used in the *StackedBarDiagram* to display its labels at the right position and with the correct value.

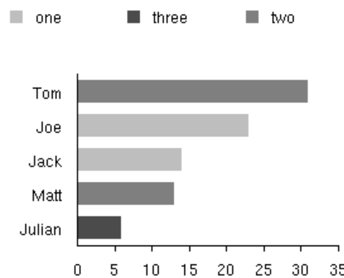
We designed the axis to have very little information about their environment. They do not know if there are any other axes or if they are at the top, bottom, left or right side of a diagram. This becomes an issue when we want to draw the labels. Depending on where the axis is they must be drawn in another direction. The diagram knows where its axes are, so we could let it tell them, in which direction the labels should be drawn. As you can see in the UML diagram of the diagrams the position of the axis can change in any subclass. For example *RangeDiagram* has its x-axis at the top while in *VerticalBarDiagram* it is at the bottom. This makes it impossible to inherit the logic for the label positioning from their common superclass (*AbstractBarDiagram*). On the other hand it leads to duplicated code if we put the logic in every diagram since there are only four places an axis can be in.

To solve this we implemented a strategy pattern. The logic for all four cases is stored in separated classes (*LeftAxisStrategy, BottomAxisStrategy, TopAxisStrategy and RightAxisStrategy*) and the diagram plugs the correct strategy into its axis by sending it the `labelPointCalculationBlock:` message with the correct strategy as parameter.

4 *EyeSee* Validation

4.1 Horizontal Bar Diagram

Horizontal bar diagrams can be useful when the order of the bars is important (for example if we have a rank order), or when the identifiers for the bars are too long and we do not want them to be rotated. Taking the example from section 2, all we have to do is change the diagram type and use the number of commits for the x axis.




```
diag := DiagramRenderer new.
(diag horizontalBarDiagram)
    identifier: #name;
    x: #numberOfCommits;
    color: #team;
    regularAxis;
    models: contributors.
diag open
```

4.2 Vertical Bar Diagram

The most common diagram is certainly the vertical bar diagram. In this kind of diagram, quantitative information is represented by bars. These can be very well distinguished because of their distinct visual appearance, making it easy to compare them and also to focus on individual bars. The base of the bars should always be zero. Even if a bar has a width and a length, only the length carries meaning. The width should be the same for all bars [2]. Therefore, in*EyeSee*, you can only define the y values as variable, the width of the bars being a constant value. We have already seen the possibilities of bar diagrams in EyeSee in section 2. Therefore, we want to put the focus on another possibility: combining diagrams.

4.3 Composite Diagrams

A powerful option in *EyeSee* is that you can combine any kind of available diagrams in one diagram. We are going to show this possibility with two vertical bar diagrams, the first one being the example from section 2. Additionally, we

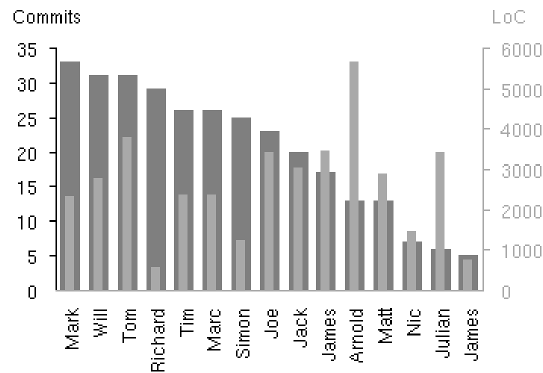
create another vertical bar diagram (Note that this could be any other diagram too), which shows us how many lines of code each contributor has written. The only difference is that we now do not have to create a DiagramRenderer, because for now, we only need the diagram, and do not want to render it yet. We add a title to each y axis with `yAxisLabel:` so we do not confuse them.

With `gapFraction` we can define, what the ratio between bar and gap is. We want to change this value from its default, so the bars which represent the lines of code are less broad than the ones representing the number of commits. Then with `rightYAxis`, we tell *EyeSee* to move the axis from this diagram to the right side. Otherwise, both axes would be on the same side. The next step is rather optional. We chose to color the axes and bars in the same shade of gray. The color from `defaultColor:` is used for the bars, while `axisColor:` is used for everything which is related to the axis (e.g. labels, axis titles, base line). By coloring the axes and bars with the same colors, we can at first glance see which axis belongs to which set of bars.

```
diag1 := (VerticalBarDiagram new)
    identifier: #name;
    setRotatedLabels;
    y: #numberOfCommits;
    yAxisLabel: 'Commits';
    regularAxis;
    models: contributors;
    yourself.
diag2 := (VerticalBarDiagram new)
    y: #loc;
    yAxisLabel: 'LoC';
    regularAxis;
    gapFraction: 2 / 3;
    rightYAxis;
    defaultColor: #lightGray;
    axisColor: #lightGray;
    models: contributors;
    yourself.
```

What is left to do, is assembling the two diagrams. For this purpose, we have to create a DiagramRender. Then, we add the two diagrams we created, and finally set the size of the diagram. We could also set the size of the diagrams when creating every one of them, but since they need to have the same size in our example, we can define the size in the composite diagram, which does the resizing for us.

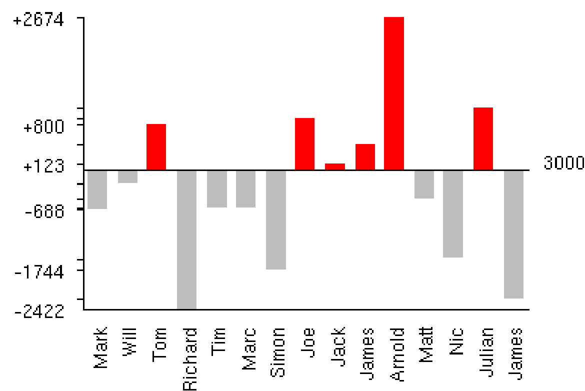
```
compositeDiagram := DiagramRenderer new.
(compositeDiagram compositeDiagram)
    addDiagram: diag1;
    addDiagram: diag2;
    height: 300;
    width: 400.
compositeDiagram open
```



4.4 Deviation Diagram

Another kind of a bar diagram is the deviation diagram. The difference to a normal bar diagram is that it does not show absolute values but the deviation from a base value. We can script this kind of diagram in *EyeSee* by providing this deviation value:

```
diag := DiagramRenderer new.
(diag deviationDiagram)
    y: #loc;
    identifier: #name;
    deviationValue: 3000;
    highlightAboveDeviation;
    valueAxis;
    models: contributors.
diag open
```



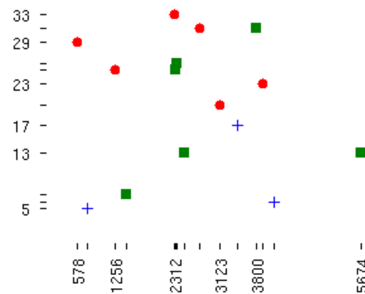
By default, all bars have the same color. However, we can choose to highlight the values which are below the deviation line or the ones which exceed the line with `highlightBelowDeviation` and `highlightAboveDeciation`. The coloring as

described of course only applies when we do not use the `color:` message to color our bars. In this case, the former messages are useless. If we do not want to display the deviation values absolutely but as percentage, we can instead use the `labelsInPercent` method.

4.5 Scatterplot

Instead of having a composite of two vertical bar diagrams, there is the possibility to visualize two attributes in one diagram with a scatterplot.

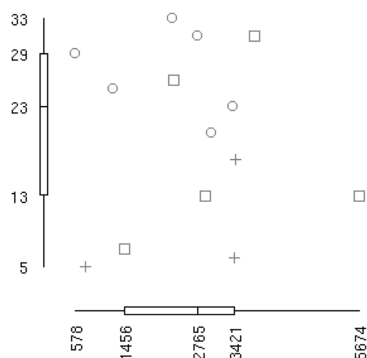
```
diag := DiagramRenderer new.
(diag scatterPlot)
    y: #numberOfCommits;
    x: #loc;
    color: #team;
    shape: #team;
    mediumObjects;
    valueAxis;
    models: contributors.
diag open
```



Dot Shapes. Besides encoding the two attributes on the y and x axis, we also encode the *team* of the contributors with the `color:` and `shape:`. If we do not encode a third attribute, we can still change the default shape (`circles`, `crosses` or `rectangles`). We can also define the size (`mediumObjects`, `smallObjects` or `bigObjects`) and the drawing style of the shapes (by default filled). Changing the drawing style to `strokedShapes` can be useful when a lot of points are close together, making it hard to tell the actual count of points in some areas. In our example, we can see this on the rectangles which overlap.

```
diag := DiagramRenderer new.
(diag scatterPlot)
    y: #numberOfCommits;
    x: #loc;
    shape: #team;
    mediumObjects;
    strokedShapes;
```

```
    boxPlotAxis;
    models: contributors.
diag open
```

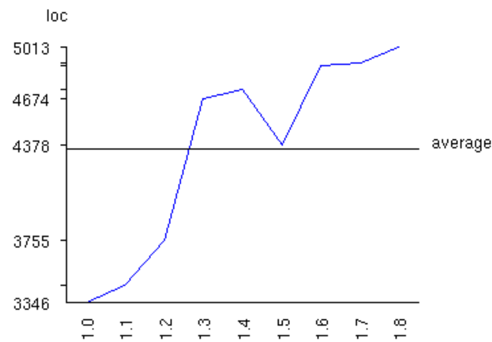


Alternative Axis. Another variation of axis is available with `boxPlotAxis`, which provides as additional information the median, upper and lower quartile and the maximum and minimum. A simpler version can be obtained with `rangeAxis`, which only displays the maximum and minimum.

4.6 Line Diagram

Usually, line diagrams are used for showing mathematical functions or if we want to display a set of data with continuous values whose order has meaning. In our example, we want a line diagram which represents the change of *lines of code* of a project. Instead of using a regular axis, we use a value axis. Notice that when there are many values which are close in the diagram, not all the labels get displayed, because they would overlap. Only the first value of a series of close values is displayed as label. But even in this case the ticks are drawn anyway so it is easy to tell if there is only one further value in the neighborhood or if there a lot.

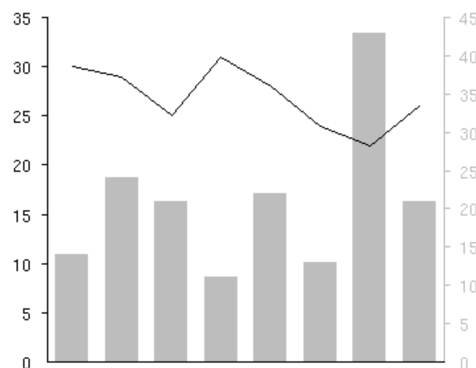
```
diag := DiagramRenderer new.
diag lineDiagram
    y: #loc;
    identifier: #versionNumber;
    yAxisLabel: 'LoC';
    valueAxis;
    defaultColor: #blue;
    models: versions;
    deviationValue:
            ((versions collect: #loc) average);
    deviationDescription: 'average';
    axisMinY: (versions minValue: #loc).
diag open.
```



Deviation lines. These can be used to emphasize a certain value. In our example it represents the average value of *lines of code* of all versions. The problem is, that a mere number cannot express its meaning by itself. By adding a description for the deviationLine, not the deviation value is displayed on the right side of the line, but the description, and we can explicitly state our intention of the deviation line.

Defining Range. Another issue is the minimum value on the y axis. For bars we stated that they should always start at zero. This does not necessarily have to be true for line diagrams. In our case, all values are between three and five-thousand. This means we have a lot of white space when our diagram starts at zero on the y axis. This can be avoided by setting the minimum value with `axisMinY:`. We can alternatively also set the range of the axis with the corresponding methods (`axisMaxX:`,...).

Mathematical Line. The line of the diagram does not start right at the y axis, as you may have noticed. This approach is rather un-mathematical, but is used when we want to compose a line diagram and a vertical bar diagram, so that the points of the line are aligned with the bars. If we nevertheless want the line to start at zero, we can use `startLineAtZero`.



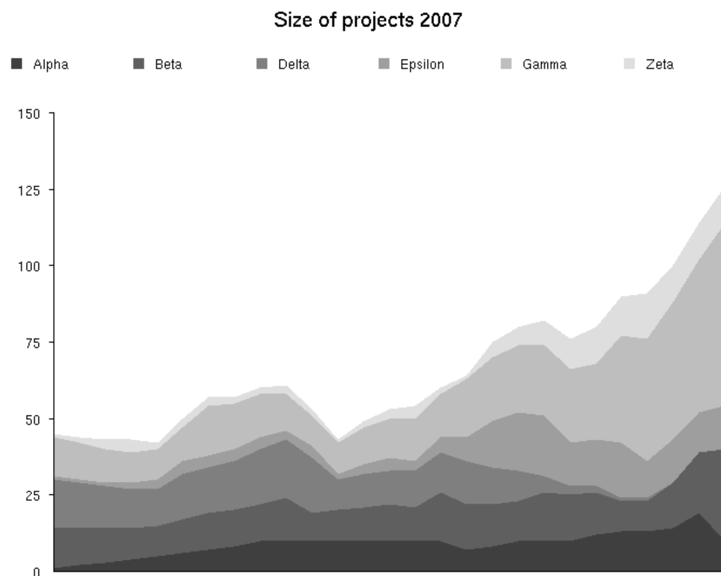
4.7 History Diagram

The history diagram is an alternative to a line diagram with multiple lines with the difference that the lines are not just drawn overlapping each other but are stacked on top of each other.

In our example, we have a collection of projects (Alpha, Beta,...), each having stored its history with the different versions. What we want to display is the change of lines of code of the versions from all projects, so we can compare the trends of the size of the projects with each other.

`dataset:` tells *EyeSee* which collections of values should be used for the different lines (here: the collection of versions of the project), and `y:` then tells it how the actual values displayed on the y axis can be resolved from these collections (lines of code of each version). For the shading of the area below the lines, we use the name of the project.

```
diag := DiagramRenderer new.
diag historyDiagram
    dataset: #versions;
    y: #linesOfCode;
    regularAxis;
    diagramTitle: 'Size of projects 2007';
    color: #projectName;
    models: projects.
diag open.
```

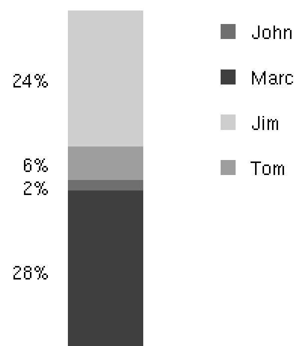


4.8 Stack Diagram

Our eye is not very good at comparing areas, especially if they are not rectangular. So using pie charts is not a very good strategy to present data. Stephen Few

suggests instead another approach for part-to-whole graphs: Stack diagrams [1]. Even if bar diagrams are still a better way to display part-to-whole data, we implemented this kind of diagram as a replacement for the pie chart.

```
diag := DiagramRenderer new.
(diag stackDiagram)
    y: #impactOnSystem;
    color: #name;
    diagramTitle: 'Percentage of Participation';
    models: contributors.
diag open
```



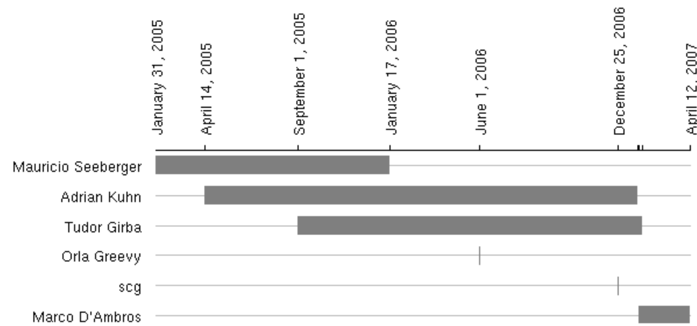
Legends. The legend in the stacked bar diagrams is automatically put on the right side of the diagram. The default placement of the legend can be scripted in every kind of diagram which supports coloring with `verticalLegend` or `horizontalLegend`.

4.9 Range Diagram

Assuming one wants to visualize the time period during which a contributor has been active in a project, we do not have an accurate way to do this with the previously presented diagrams. One solution to do this is using a range diagram, which uses bars to show activity.

```
(diag rangeDiagram)
        identifier: #name;
        minX: #firstVersionTime;
        maxX: #lastVersionTime;
        labels: #asDate;
        valueAxis;
        models: contributors.
diag setup.
```

Because the bars do not start at zero, we have to provide two values for each bar. In our example, we get the time in seconds of the first version a contributor



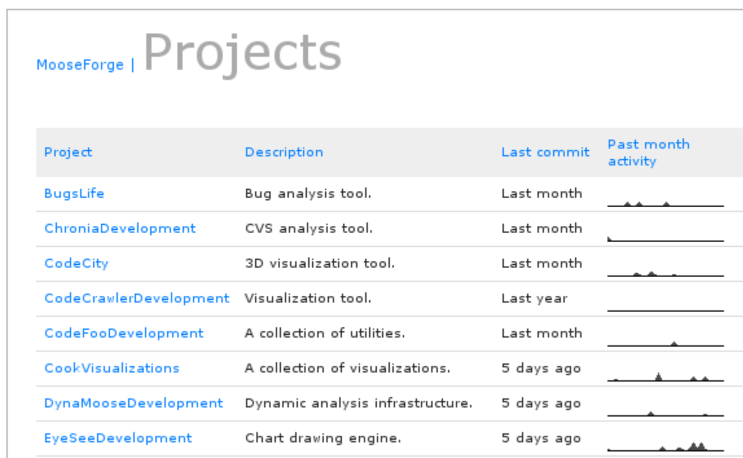
committed for the beginning of the bar, and the corresponding value of the last version, and tell *EyeSee* to use these values with `minX:` and `maxX:`.

Formatting Labels. Let us assume that the point of time is provided by a number of seconds since a determined point of time. If we use this number of seconds for the labels on the axis, no human being could possibly get any meaning from them. If we want to change the manner in which the labels are displayed, we can use `labels:` to define a block how the labels get formatted. In our example, we use this technique to change the provided label value in seconds to a human-readable date.

5 Field study and future work

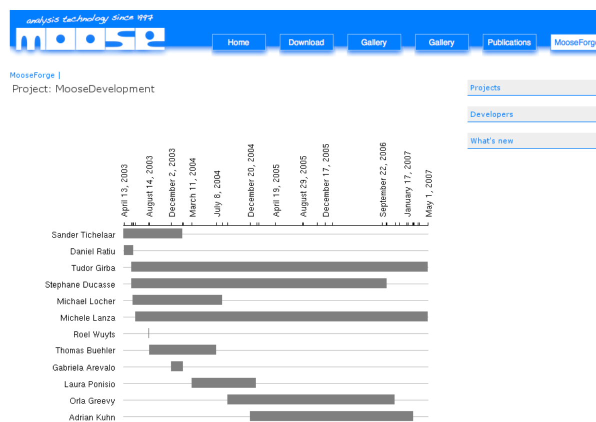
5.1 MooseDen

MooseDen is a Seaside project which among other things provides in the Moose-Forge section information about different Smalltalk projects developed at the University of Bern. To display the diagrams on websites, we use *EyeSee*'s ability to export the diagrams as png images. The first idea was to visualize activity of different projects in the past month. Because this diagram was supposed to be placed within a table, the space available for the diagram was rather small. By setting the padding to zero, and not adding any labels or axis, we made sure to maximize the space used for the data. And instead of using a line diagram, we used a history diagram with only one line.



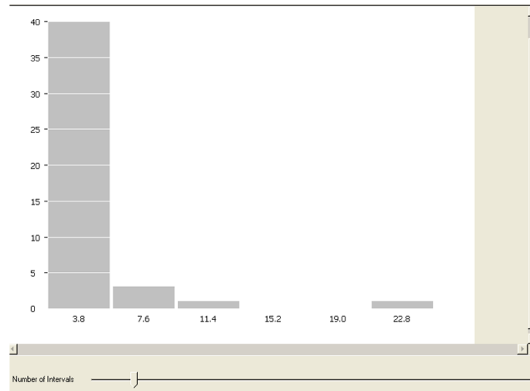
Project	Description	Last commit	Past month activity
BugsLife	Bug analysis tool.	Last month	
ChroniaDevelopment	CVS analysis tool.	Last month	
CodeCity	3D visualization tool.	Last month	
CodeCrawlerDevelopment	Visualization tool.	Last year	
CodeFooDevelopment	A collection of utilities.	Last month	
CookVisualizations	A collection of visualizations.	5 days ago	
DynaMooseDevelopment	Dynamic analysis infrastructure.	5 days ago	
EyeSeeDevelopment	Chart drawing engine.	5 days ago	

The second application was to display the period of activity of the contributors for each project. The solution was a range diagram.

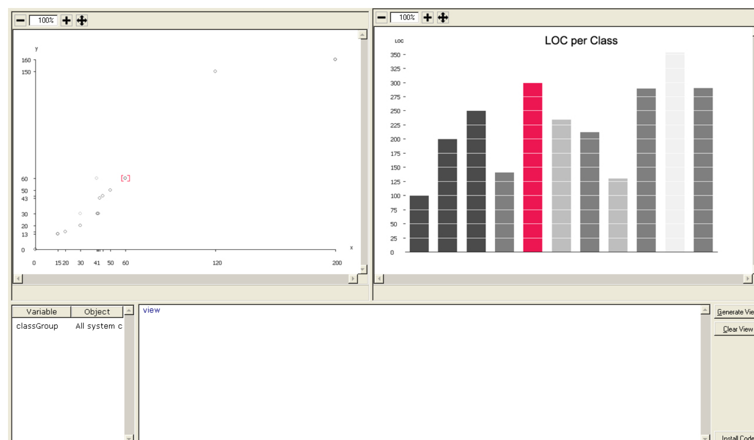


5.2 Interactivity

To make the customization of diagrams a bit easier we plan to build a graphical user interface for *EyeSee*. A benefit of this for the user is that he does not have to memorize or look up the name of the scripting commands. Instead he can change the parameters directly via sliders, checkboxes etc. For example the number of intervals in a histogram can be changed using a slider.



Another direction we want to advance *EyeSee* in is interaction between diagrams. For example there often are datasets that have more than two or three relevant values. To address this, we could use a third attribute like shape or color or even introduce a third dimension in our diagram. The problem with this approach is that it makes the diagrams harder to read and a three-dimensional chart does often not convey its message well on a two dimensional medium. To solve this it might be possible to create a visualization that shows two diagrams at the same time. For example a scatterplot and a bar diagram. If the user selects a point in the scatterplot the corresponding bar in the bar diagram would be highlighted. Or maybe we want to display a popup containing a small histogram when we move the mouse over a property to see how it is distributed. We want such interactions to be scriptable directly within the tool that is used to create the diagrams.



6 Conclusion

Many diagram drawing tools require this input to be passed in a fixed format. These tools are not flexible enough to be adapted to different models. Furthermore the output of these tools is often a diagram that is cluttered with chart junk that distracts the reader from the content. The goal of *EyeSee* was to create a diagram drawing engine that addresses these problems. We implemented scripting methods to make it easy to create and customize diagrams. The scripts are small and hide the internals of our model. We also provided a scripting interface to allow the user to tell *EyeSee* how to extract data from the used model. This makes *EyeSee* model-independent. Finding the diagram that is most suitable for communicating the given set of data can be done quickly, because *EyeSee* gives the user a wide range of predefined solutions. However, *EyeSee* allows the user to customize a lot parameters (for a complete list, see the appendix). The user cannot only customize the diagrams, but also compose them in any way he wishes to.

We validated *EyeSee* by drawing some of the most common diagrams using short scripts and by using *EyeSee* in MooseDen we showed how easy it is to draw diagrams with any kind of model.

Our plans for the future of *EyeSee* are to provide a graphical user interface to make it more user-friendly. We will also take the opportunity of the user interface to explore the possibilities of interactivity. Besides, we will also look for new kinds of diagrams we could implement to expand the possible range of application.

7 Appendix

7.1 Quick Reference

In this section, you can find a complete list of all diagrams with the available scripting methods. An ‘x’ means, that the diagram understands the scripting method, while a ‘-’ means, that it does not or the combination would not make sense.

	Vertical Bar	Horizontal Bar	Composite	Deviation	Scatterplot	Line	History	Stack	Range
`height:`	x	x	x	x	x	x	x	x	x
`width:`	x	x	x	x	x	x	x	x	x
`xPadding:`	x	x	x	x	x	x	x	x	x
`yPadding:`	x	x	x	x	x	x	x	x	x
`axisMaxX:`	-	x	-	-	x	-	-	-	x
`axisMinX:`	-	-	-	-	x	-	-	-	x
`axisMaxY:`	x	-	-	x	x	x	x	-	-
`axisMinY:`	-	-	-	-	x	x	-	-	-
`models:`	x	x	-	x	x	x	x	x	x
`x:`	-	x	-	-	x	x	x	x	-
`y:`	x	-	-	x	x	x	x	x	-
`minX:`	-	-	-	-	-	-	-	-	x
`maxX:`	-	-	-	-	-	-	-	-	x
`identifier:`	x	x	-	x	-	x	-	-	x
`color:`	x	x	-	x	x	-	x	x	-
`defaultColor:`	x	x	-	x	x	x	-	-	x
`axisColor:`	x	x	-	x	x	x	x	x	x
`verticalLegend`	x	x	-	x	x	-	x	x	-
`horizontalLegend`	x	x	-	x	x	-	x	x	-
`useColors:`	x	x	-	x	x	-	x	x	-

	Vertical Bar	Horizontal Bar	Composite	Deviation	Scatterplot	Line	History	Stack	Range
`setInvertedLinearFill`	x	x	-	x	-	-	-	x	-
`setLinearFill`	x	x	-	x	-	-	-	x	-
`setColoredFill`	x	x	-	x	-	-	-	x	
`diagramTitle:`	x	x	-	x	x	x	x	x	x
`xAxisLabel:`	-	x	-	-	x	-	-	-	x
`yAxisLabel:`	x	-	-	x	x	x	x	-	-
`labels:`	x	x	-	x	x	x	x	x	x
`setRotatedLabels`	x	x	-	x	x	x	-	-	x
`regularAxis`	x	x	-	x	x	x	x	-	x
`valueAxis`	x	x	-	x	x	x	x	-	x
`boxPlotAxis`	x	x	-	x	x	x	x	-	x
`rangeAxis`	x	x	-	x	x	x	x	-	x
`rightYAxis`	x	-	-	x	-	x	-	-	-
`gridLine`	x	x	-	x	-	-	x	-	-
`deviationValue:`	x	x	-	x	-	x	-	-	x
`deviationDescription:`	x	x	-	x	-	x	-	-	x
`highlightAboveDeviation`	-	-	-	x	-	-	-	-	-
`highligthBelowDeviation`	-	-	-	x	-	-	-	-	-
`shape:`	-	-	-	-	x	-	-	-	-
`smallObjects`	-	-	-	-	x	-	-	-	-
`mediumObjects`	-	-	-	-	x	-	-	-	-
`bigObjects`	-	-	-	-	x	-	-	-	-
`strokedShapes`	-	-	-	-	x	-	-	-	-
`circles`	-	-	-	-	x	-	-	-	-
`crosses`	-	-	-	-	x	-	-	-	-
`rectangles`	-	-	-	-	x	-	-	-	-
`addDiagram:`	-	-	x	-	-	-	-	-	-
`dataset:`	-	-	-	-	-	-	x	-	-
`gapFraction:`	x	x	-	x	-	-	-	-	x

7.2 Availability

EyeSee can be obtained via Cincoms VisualWorks (http://smalltalk.cincom.com/) in the store of the SCG group at the University of Bern by logging in with:

- Environment: db.iam.unibe.ch:5432_scgStore
- User Name: storeguest
- Password: storeguest

If you load the latest version of the bundle *EyeSeeDevelopment*, you can find all the examples provided in this paper and more in the *EyeSeeTest* package in the *Examples* class. There you can also find the models we used in the protocol *models*.

You can run these examples by executing the first line in each example which is commented out (for example `"self new lineDiagram"` in the lineDiagram example).

References

[1] Stephen Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[2] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[3] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.