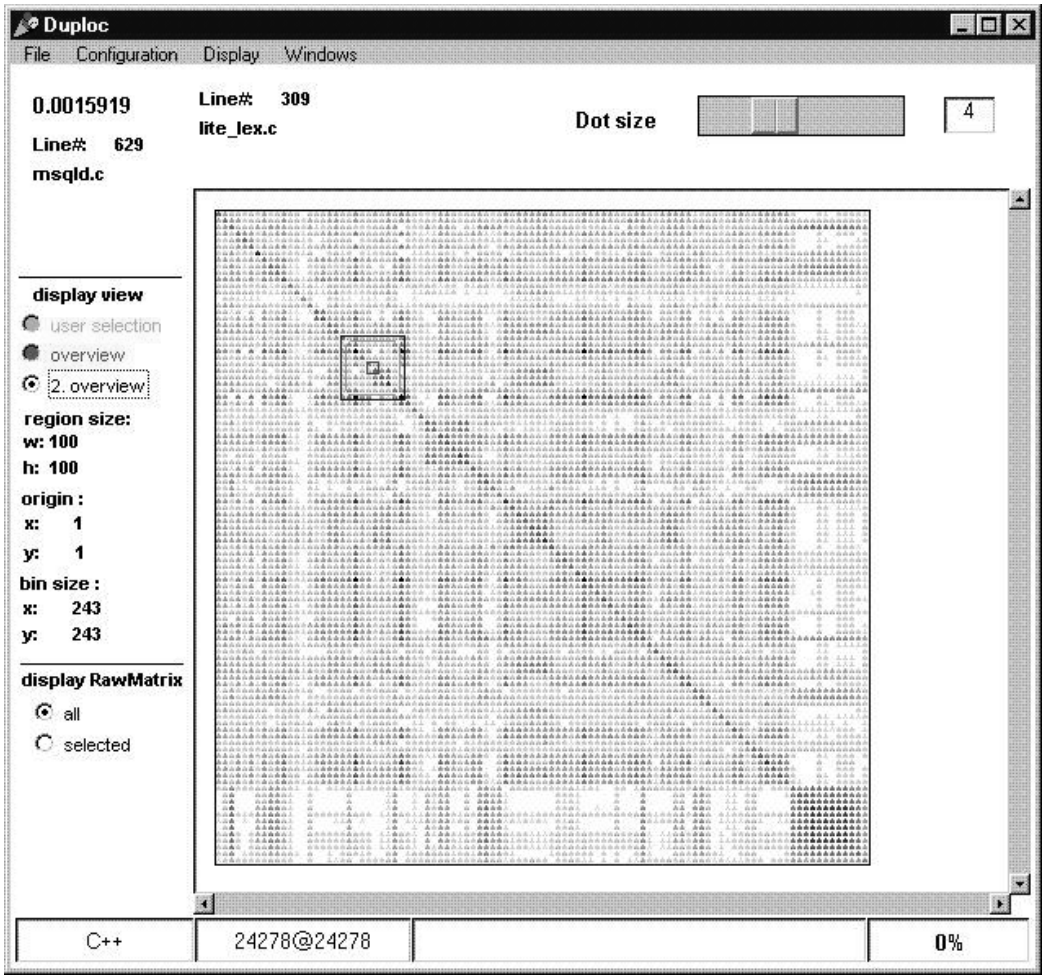
computer science project 98/99

“Information Mural“ visualisation of Duploc

Pietro Malorgio
Software Composition Group, University of Berne
malorgio@iam.unibe.ch

Wednesday, 14. July 1999



Contents

1 Project summary

*Duplo*c is a tool written in Smalltalk (VisualWorks 2.5/3.0), which is currently under continuous development inside the *Software Composition Group* at the *University of Bern*[1]. It is designed for representing graphically the comparison results of found duplicated lines of code (duploc) out of a set of loaded source code files. *Duploc* supports different programming languages (C++, C, Java, Smalltalk etc.). The loaded files are compared line-by-line using a simple string-match comparison function – the comparison results are stored in a two dimensional *comparison matrix*. The previous *Graphical User Interface* (*GUI*) represents the obtained *comparison matrix* as a *dotplot* diagram – in this two dimensional grid of black painted dots, each dot stands for two identical found lines of code in two different files. Figure 1 shows a *dotplot* diagram of a *comparison matrix* with 229x229 elements. (The *comparison matrix* size is shown with the format '*height@width*' in the bottom information line of the *GUI*.)

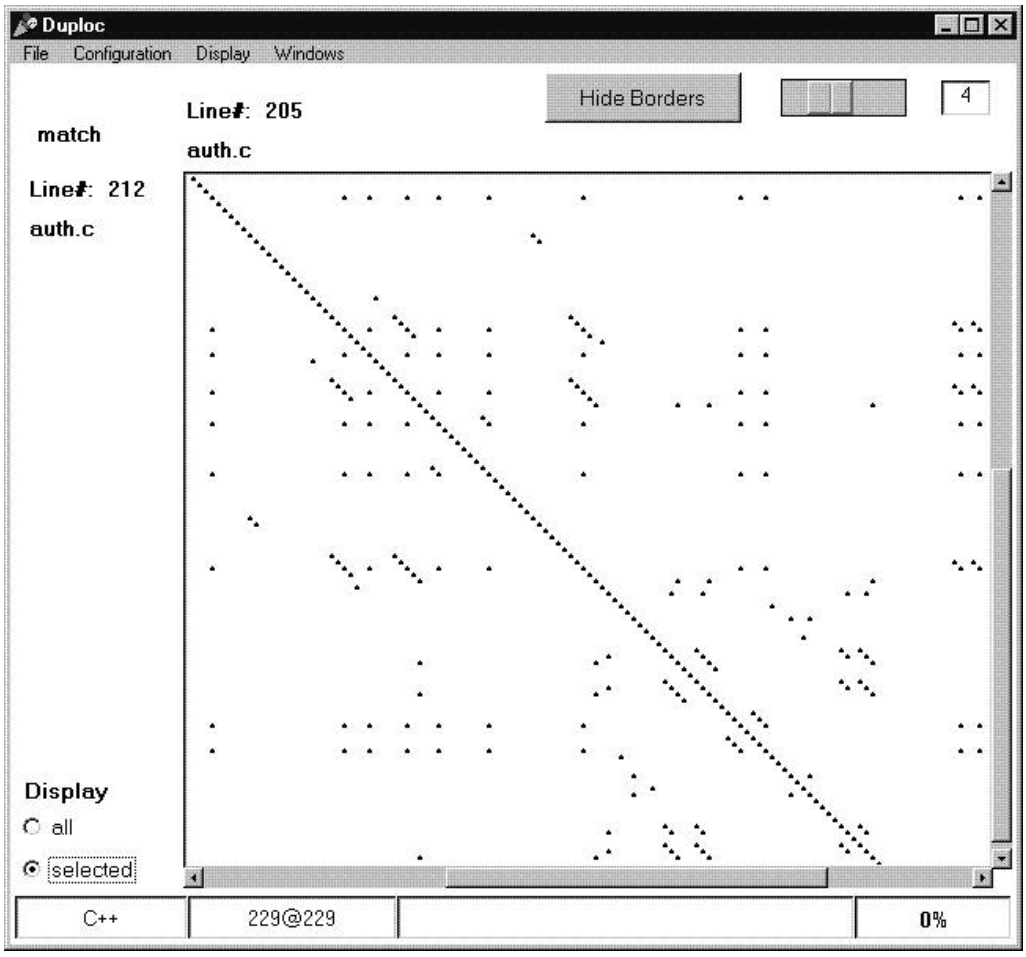


Figure 1: The previous implemented GUI.

[1] http://www.iam.unibe.ch/~rieger/duploc/index.html

This *GUI* uses a scrollbar to provide some navigation facility over the *comparison matrix*. It is therefore only suitable for visualising *comparison matrixes* up to some hundred elements per matrix side (e.g. 800x800). The project goal was to integrate into the *Duploc* application a technique named *Information Mural* in order to visualise a large *comparison matrix*. Figure 2 shows the *Information Mural* overview image of a *comparison matrix* with 24278x24278 elements. This image was produced with the new developed *GUI*. Each dot stands for the 'match density' inside a correspondent region in the underlying *comparison matrix*. Darker dots indicates a region of the *comparison matrix* with more matches then lighter dots. This new developed *GUI* is typically capable to visualise a *comparison matrix* with up to two million elements per side. It also provides navigation facilities for exploring parts of the *comparison matrix* in a *dotplot* like display mode. This display mode appears like the previous *GUI* - see Figure 1.

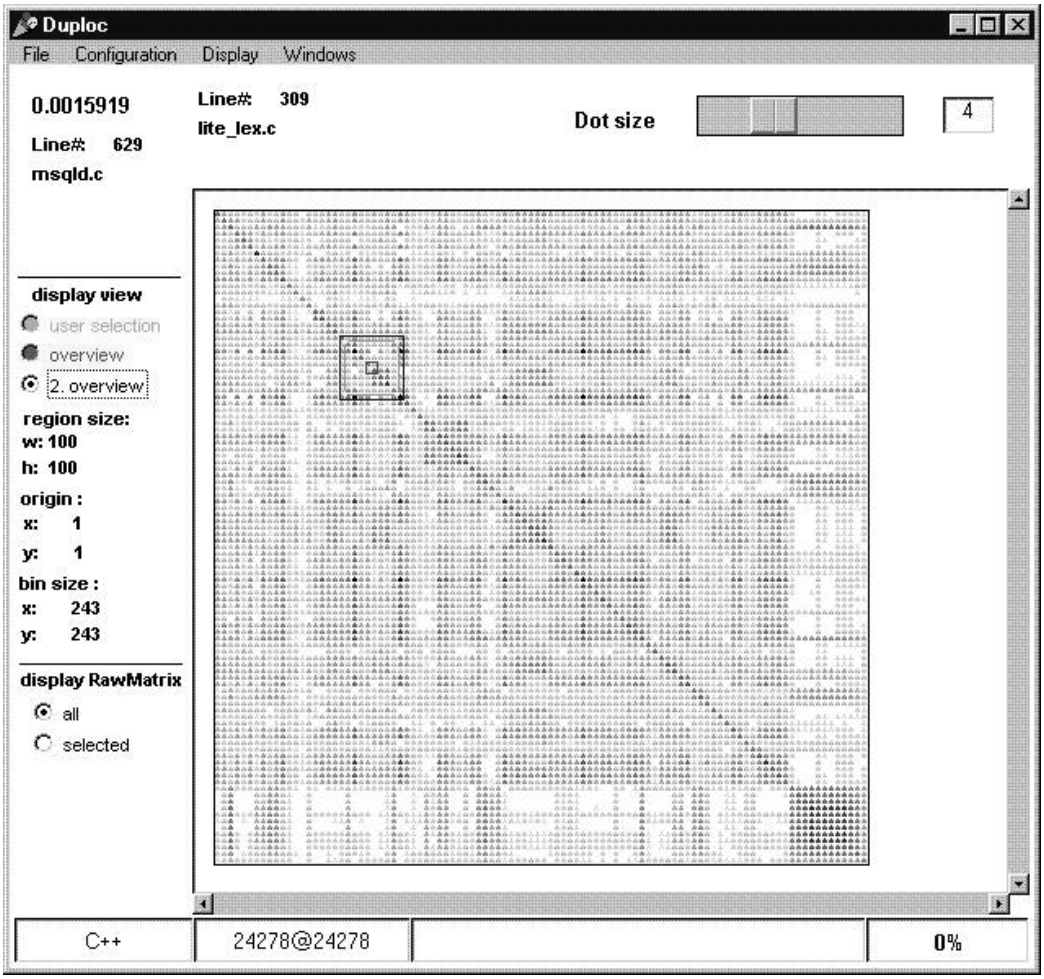


Figure 2: The new developed GUI.

Project context

This project was formulated as the mandatory project in the computer science course. *Pietro Malorgio*, the author, wrote it under the supervision of *Matthias Rieger* and *Dr. Stephane Ducasse*.

Structure of this document

This document is divided in 6 main chapters. Chapter 2 gives from an user point of view an introduction to some comparison aspects and to the previous *Duploc* application. Chapter 3 presents the *Information Mural* technique, which was used in the new developed *Graphical User Interface* presented in Chapter 4. The internal application design is the subject of Chapter 5. The last chapter discusses the conclusions drawn from the project.

Acknowledgement

I would like the express my thanks first to Stéphane for having introduced me to this project, then to Matthias, with which I spent some many hours talking about the project and who implemented the discussed changes to his previous work in order to implement these achieved extensions. My thanks goes also to Prof. Nierstrasz for having set early the guidelines of this project, to Serge Demeyer who helped to keep the project on track and again to Stéphane, who gave me some early introductions to the Smalltalk environment. For the received feedback about the user interface I can not forget to express my thanks to Sander Tichelaar and Robb Nebbe.

Pietro Malorgio

2 An introduction to Duploc

2.1 Dotplot - a technique for representing duplicated lines of code

The purpose of *Duploc* is to identify duplicated lines of code. This document will not describe the actual implemented comparison process for finding duplicated lines of code – instead it describes, how duplicated lines of code are represented. ***Dotplot***[1] is a technique for visualising patterns of string matches in millions of lines of digital information. Figure 3 shows an illustrative example of this technique by comparing words in a sentence. A sequence is tokenised and plotted from left to right and top to bottom with a dot where the tokens match. Dots off the main diagonal indicate similarities. This *dotplot* is symmetrical, because the two compared sentences are identical.

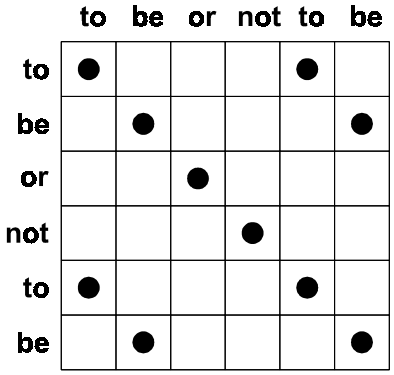


Figure 3: Six words of Shakespeare.

The information in a *dotplot* can be represented as a matrix of Boolean like data type – this type of matrix is referenced in this document as a ***comparison matrix***. A *dotplot* is a graphical representation of a *comparison matrix*. The example above is extended in Figure 4 by adding another sentence, probably appropriate for our computer science time period: 'to copy or not to copy'.

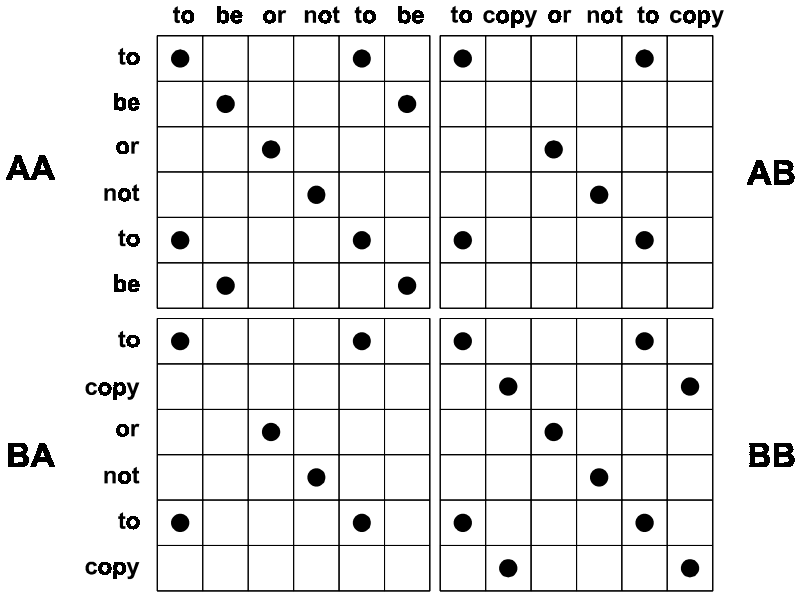


Figure 4: Comparing two sentences.

There are four *comparison matrixes* for these two sentences. Let's reference them in the following way. 'A' stands for the first sentence: "to be or not to be". 'B' stands for the second introduced sentence: "to copy or not to copy". Therefore the upper left *comparison matrix* is referenced with 'AA' (= 'A' vs. 'A') and the bottom right *comparison matrix* is referenced with 'BB'. The bottom left *comparison matrix* is referenced with 'BA' and the top right *comparison matrix* is referenced with 'AB'. 'AA' and 'BB' are always symmetrical. They show the duplicated words in each respective sentence. In this example 'AB' and 'BA' are symmetrical, but by choosing the sentence 'B' with more or less words then the sentence 'A' they would be asymmetrical. Further, each one is the transposed *comparison matrix* of the other one ('AB' = 'BA'^{t}).
Comparing two source code files supposes to define, what has to be compared. *Duploc* compares two files line-by-line using a simple string-match comparison function. Previously, comments and white spaces are removed, so that each line is in some kind of *normal form*. Empty lines or lines containing only a '}' (in C++) are dismissed. Table. 1 shows the text of two simple C programs. The text underlined with a wave line indicates the actual text, which would be used in the comparison process of *Duploc*. Figure 5 shows the corresponding *comparison matrix*.
For a extensive interpretation of those *dotplot* patterns see paper of the Helfman [1].

UINT.C	***UINT32.C***

`#include <stdio.h>`
`#include <sys/types.h>`
 	
`main()`
`{`
`  u_int foo;`
 	
`  foo = 1;`
`}`
	`#include <stdio.h>`
	`#include <sys/types.h>`
	`#include <unistd.h>`
	
	`main()`
	`{`
	`  u_int32_t foo;`
	
	`  foo = 1;`
	`}`

Table 1. contents of file *UINT.C* and *UINT32.C*

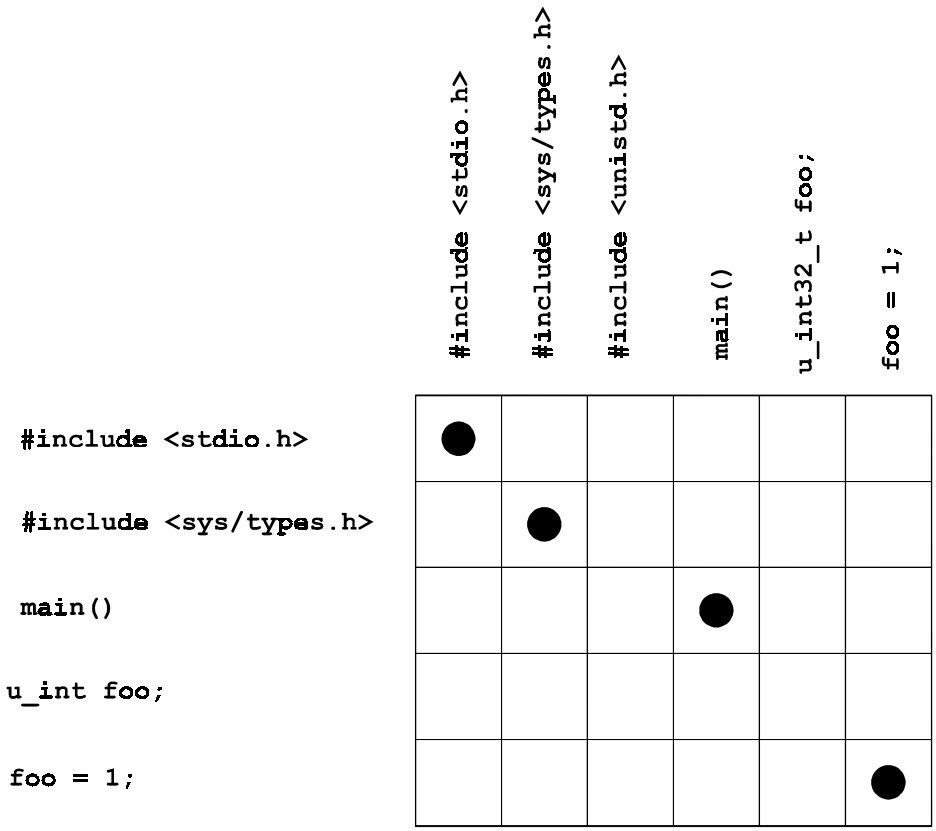


Figure 5: *Comparison matrix* of file '*UINT.C*' vs. '*UINT32.C*'.

2.2 The dataflow concept inside Duploc

The following Figure 6 illustrates the data-flow concept inside *Duploc* around the two main repositories.

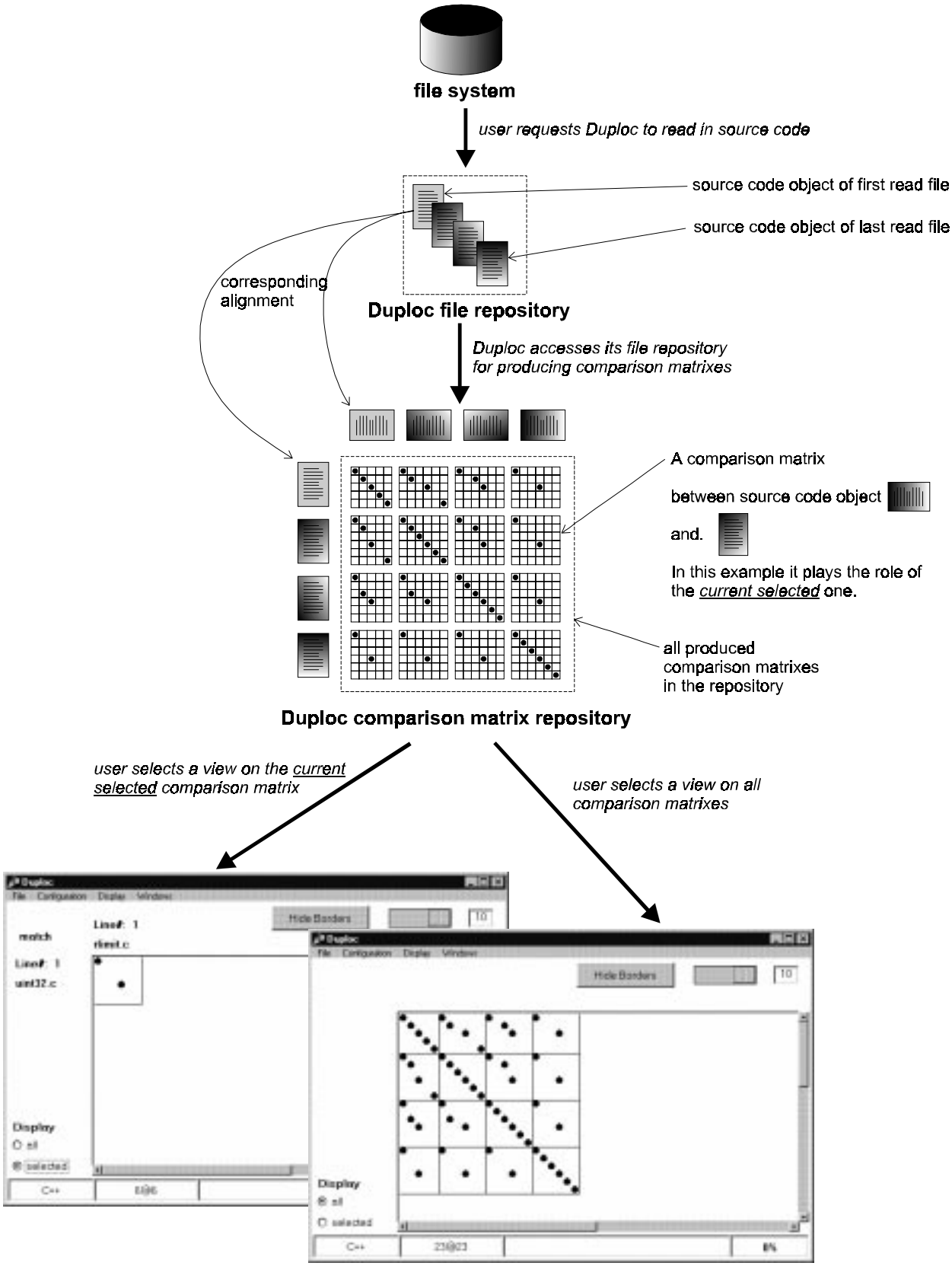


Figure 6: Data flow concept inside *Duploc*

The user requests *Duploc* to read in source code files from the underlying file system. Each read in file is transformed in a *normal form*[2] and stored in the internal *file repository*. After the transformation, the file is called a *Source Code object*. Based on the current *file repository* contents all possible combinations of *comparison matrixes* between two *Source Code object*s are stored in the internal *comparison matrix repository*. Figure 6 shows a representative image of this *comparison matrix repository*: All *comparison matrixes* form a sort of table – the alignment of the *Source Code object*s in the *file repository* influences the structure of this table, so that new read in files make the table grow in the direction of the lower right corner. The user can select two available views onto the *comparison matrix repository*:

- *all*: Displays all the *comparison matrixes* at once. The rectangular lines delimit each *comparison matrix*.

- *selected*: This mode displays the *comparison matrixes* for two selected files. The two files are selected in a auxiliary window, which lists up[3] the available files in the *file repository*.

In both views a *dotplot* of the selected *comparison matrix*(*es*) is displayed: Each match is depicted as a black painted dot with separation lines between neighbouring *comparison matrixes*. The *Graphical User Interface* provides functionality's to the user to examine in a point-and-click concept the displayed *comparison matrix*(*es*). The next section presents these available functionality's.

2.3 Using Duploc in the original Interactive Mode

The following sections present the original Interactive Mode of *Duploc*:

2.3.1 Starting the original version

Consult the '*Duploc Tutorial*' document [2] for a proper installation of the tool. The original *Duploc* version is started by evaluating

```
DuplocApplication open
```

in a VISUALWORKS workspace or by selecting the `windowSpec` resource of `DuplocApplication` in a Resource Finder and pressing START.

2.3.2 Selecting the Source Code Language

Duploc is able to read source code of a number of different languages. The code reader is set initially to the language specified in the `ini`-File[4]. The language can be changed interactively with the Select Source Language... menu item in the Configuration menu. Changing the source language takes effect immediately. This means that the next file that is read in after a language switch is assumed to be written in the newly selected language. The files already loaded will not change their language and the comparison process is language independent. (This means that you can compare C ++ files with SMALLTALK files, for a lark.)

2.3.3 Reading Source Code

Files can be read into the *file repository* using the normal VISUALWORKS **FileBrowser**, which is invoked from the File menu (see Figure 7).

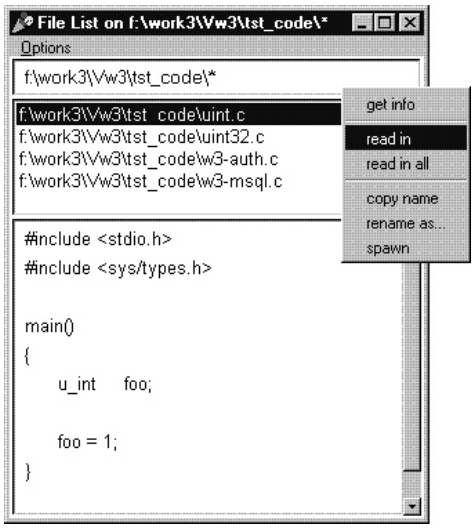


Figure 7: The *Duploc* file browser with the operate pop-up over the file list.

[2] see section above
[3] see following section
[4] consult the '*Duploc Tutorial*' document [2]

The directory path and file pattern must be entered in the top input field. The pop-up menu, appearing when the operate button[5] is pressed on the file list in the middle of the window, offers two items which are *Duploc* specific:

1. *read in*: Reads in the selected file and puts it into the *file repository*.

2. *read in all*: Reads all the files that are currently displayed in the file list and puts them into the *file repository*.

The files that are currently loaded into the tool are displayed in the window that is opened with Windows >> File List (see Figure 8). The operate pop-up menu for this list allows to remove individual files from the *file repository*. The same window is also used for determining the currently defined *'selected' comparison matrix* (see next section).

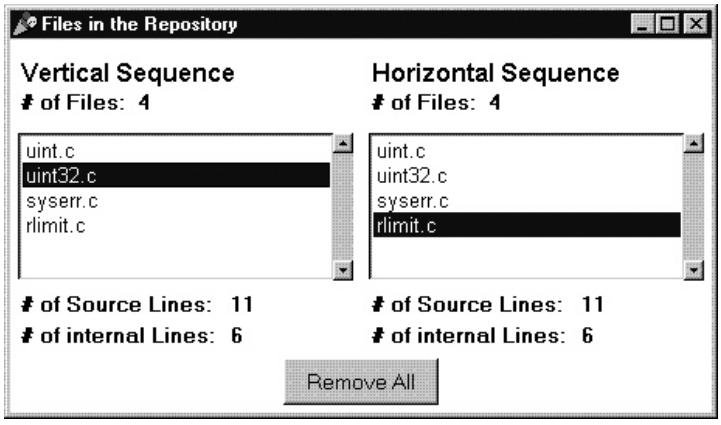


Figure 8: The *Duploc file repository* contents.

2.3.4 Viewing the comparison matrix(es)

Once files are present in the *file repository* two views are available onto the *comparison matrix repository*, which contains all *comparison matrixes*. The selection occurs with the radio button group on the lower left side of the main window labelled with 'Display' – see Figure 9.:

- *all*: Displays all the *comparison matrixes* at once (see right window in Figure 9). The rectangular lines delimit each *comparison matrix*.

- *selected*: This mode displays the *comparison matrix* for two selected files (see left window in Figure 9). The two files are selected in the auxiliary window, which lists up the available files in the *file repository*. (see Figure 8. – currently the *comparison matrix* between file '`uint32.c`' and '`rlimit.c`' is displayed.)

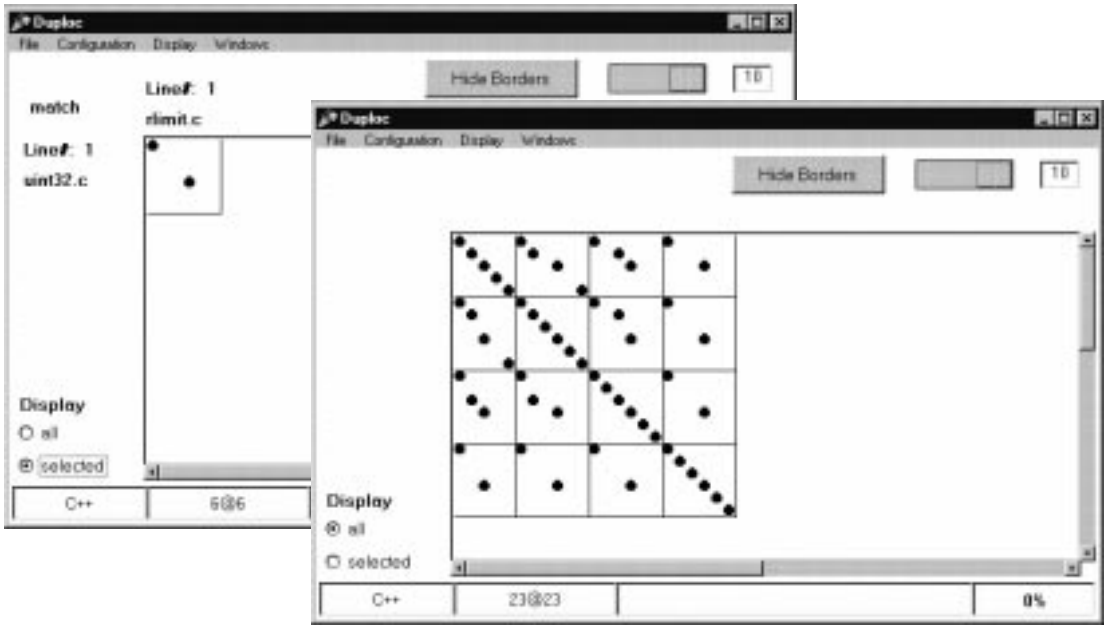


Figure 9: The two available views: '*all*' vs. '*selected*'.

[5] The operate is the middle mouse button

The bottom information line in the main window displays following information:

- *First field*: The current selected source language.
- *Second field*: The total vertical and horizontal size of all currently displayed *comparison matrixes*. (see right window in Figure 9 – it indicates, that the size of all *comparison matrixes* is currently 23x23.)
- *Third field*: This field shows during the comparison process, which *comparison matrix* is currently built.
- *Fourth field*: This field shows the percentage progression of a comparison process in progress (N.B. Figure 9 shows '0%', because no comparison process is in progress) .

2.3.5 Exploring the selected comparison matrix(es)

Once a view is selected, following functions allow the user to examine the *comparison matrix(es)*:

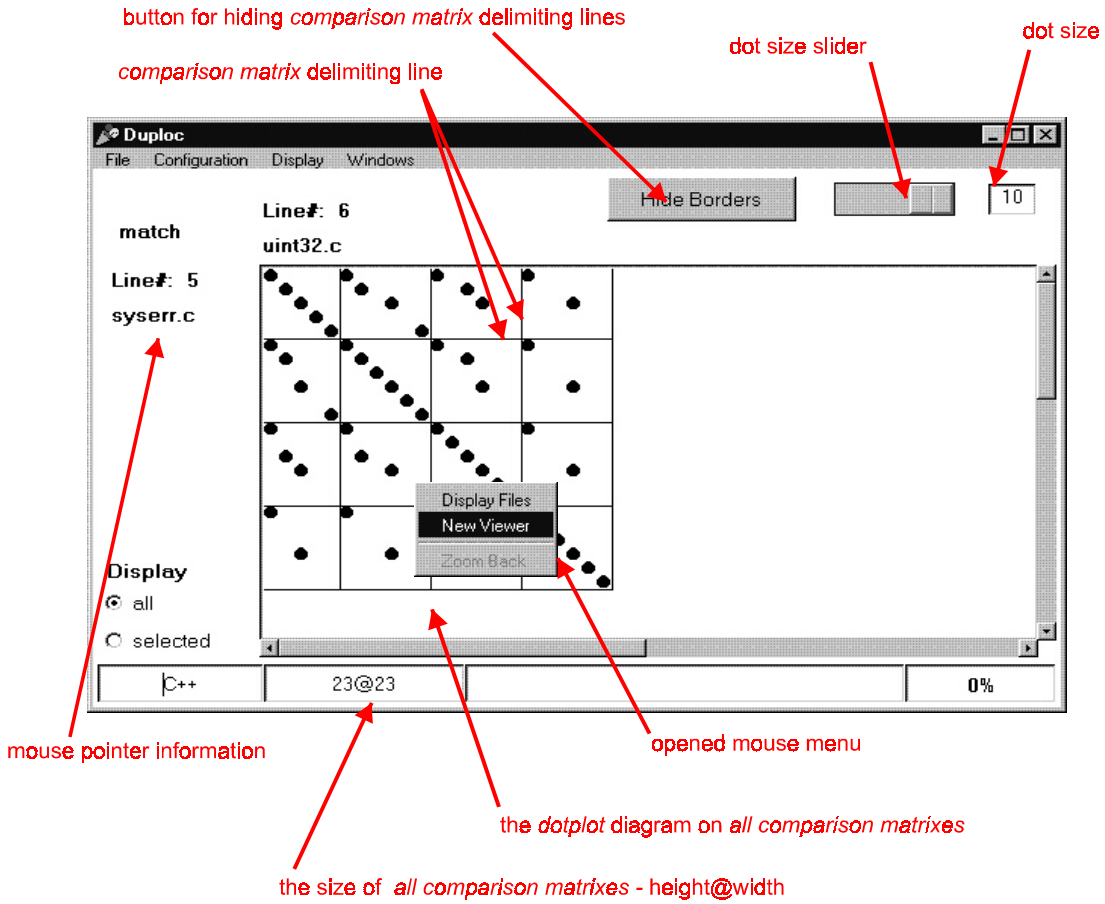


Figure 10: The main window with the opened mouse menu.

Dot size slider

The size of the dots can be selected between 1 and 10 screen pixels with the slider in the upper right corner of the main window – see Figure 10.

Hiding Borders

The delimiting lines between neighbouring *comparison matrixes* can be hidden by pressing the button Hide Borders or by deselecting the menu item Show Borders in the Display window menu – see Figure 10. The reverse of this function can be obtained by pressing on the same button respectively by selecting the same menu item.

Zooming

A zoom facility allows to enlarge interesting zones of the *dotplot*. By pressing the left mouse button over the *dotplot diagram*, while dragging the mouse, a rubber band rectangle defines the zone which should be enlarged. By selecting Zoom Back on the mouse menu, the zooming can be undone – see disabled menu item below the menu item NewViewer in the opened mouse menu in Figure 10.

Source Code Examination

By using the mouse cursor on the *dotplot*, the user can explore the compared files. The co-ordinates of the mouse cursor translated to a position in the source code is displayed in the upper left corner of the main diagram – in Figure 10 the mouse cursor is not visible, but its current position is over the dot, which stands for the found match between line 5 of

file ‘`syserr.c`’ and line 6 of file ‘`uint32.c`’. By clicking with the left mouse button on a dot the user opens a ‘*Files compared*’ window which displays the source code of the two compared files. The lines that matched are emphasised in red (see Figure 11). By opening the mouse menu with the operate button on the same dot and selecting the menu item Display Files, the equivalent action is achieved.

Filtering

When looking at matched source code lines, the user may decide that specific lines are not interesting. By clicking on the Delete Line button of the ‘*Files compared*’ window, the emphasised line is marked as ’deleted’. All the matches that are generated by these lines are no longer displayed. The user can look at all the lines that are currently ’deleted’ using the window that is opened in the main window under Window >> Deleted Lines. Individual lines can be ’undeleted’ through the operate mouse button pop-up menu.

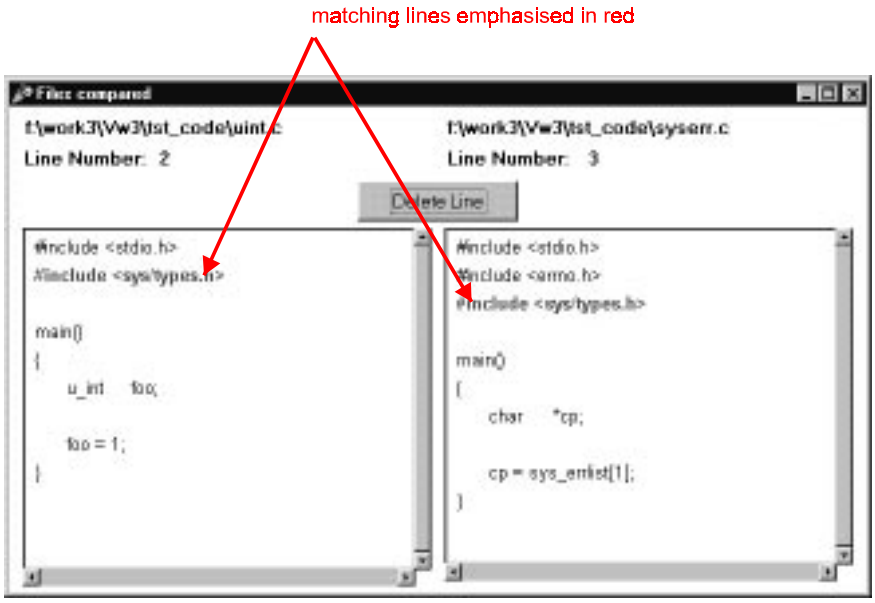


Figure 11: A opened 'Files compared' window.

3 The project goal

3.1 The problem – a limited display capability

The visual representation of large quantities of *comparison matrix(es)* is subject to a 'natural' barrier, which is the number of pixels on the screen. The previous implementation of *Duploc* makes it possible to display a *dotplot* down to a maximal reduction of one pixel per dot. This limits the quantity of *comparison matrix(es)* presented with a *dotplot* in a reasonable way onto the screen. Depending on the normalised file sizes, a *dotplot* with some hundred rows respectively columns can be displayed. e.g. a *dotplot* with 800x800 elements needs at least 800x800 screen pixels. This screen pixel number increases to 8000x8000, if the maximal dot size of 10 pixels per dot is selected. Such a *dotplot*, which can cover between 800x800 and 8000x8000 pixels, can still be explored with two scrollbars. A *dotplot* with several thousand rows respectively columns can not be explored in a reasonable way anymore. Even by providing a scrollbar the overall context will be lost.

3.2 The solution - overcoming this limitation with the Information Mural technique

The technique of the *Information Mural* enables to visualise several data elements per pixel [JS96]. The *Information Mural* is a two-dimensional, reduced representation of an entire information space that fits entirely within a display window or screen. The mural creates a miniature version of the information space using visual attributes such as greyscale shading, intensity, colour, and pixel size, along with anti-aliased compression technique. The original information space is partitioned in equal sized regions – named *bin regions*. Each *Information Mural* element, named *bin*, represents some characteristics of the correspondent *bin region*. Consult the paper [JS96] for the different presented applications.

In the context of this project, the information space that must be represented is a matrix[6] of Boolean like data type – see in Figure 12 the upper left *dotplot* representing this matrix. This matrix is partitioned in equal sized 2x2 rectangles, referenced as the *bin regions* – see in Figure 12 the bottom left *dotplot*.

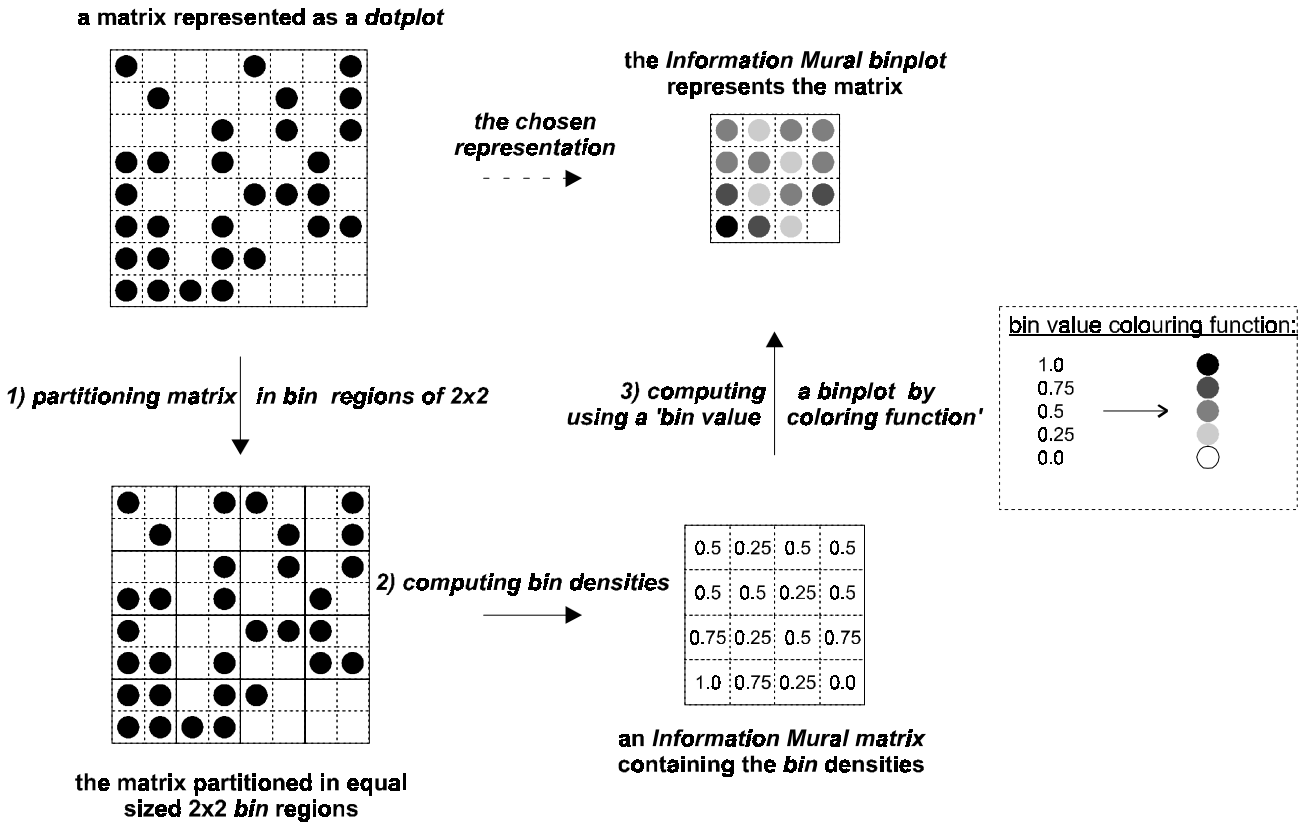


Figure 12: The *Information Mural* concept.

The chosen representation of the observed matrix is its match density: An *Information Mural matrix* is computed, where each matrix element represents the match density (*bin* density) inside the correspondent *bin* region – see the bottom

[6] N.B. no reference to a *comparison matrix* is made here. See further down the introduced term *raw matrix*.

right matrix of floats in Figure 12. The *Information Mural* is the graphical representation of this *Information Mural matrix*. By using a *bin value colouring function* each *Information Mural matrix* element is represented in the *Information Mural* as a grey shaded dot. The obtained plot is referenced as the *binplot* – see the top right plot in Figure 12. In the *binplot* of Figure 12 a darker dot stands for a denser and a lighter dot for a sparser *bin* density inside the correspondent *bin* region.

3.3 The project goal – integrating the Information Mural technique into Duploc

The project goal was to integrate the *Information Mural* technique into the *Graphical User Interface* of *Duploc* by:

- providing an *Information Mural overview binplot* of the current selected *comparison matrix(es)*.
- providing navigation means, in order to explore sections of the *comparison matrix(es)* with a conventional *dotplot*.

4 The new Graphical User Interface

4.1 Newly introduced concepts

4.1.1 The *raw matrix set*

The first dataflow concept (see Figure 6) was extended by the *raw matrix set* – see Figure 13. The reason for the introduction the concept of the *raw matrix set* is, that each defined *raw matrix* groups the contents of a specified set of *comparison matrixes* and presents them as a single large matrix:

- *Definition*: A *raw matrix set* contains some defined *raw matrixes*.
- *Definition*: A *raw matrix* groups the contents of a defined set of *comparison matrixes*. It has a co-ordinate system, where the top left element has co-ordinates (1,1).

Currently two *raw matrixes* are defined; they correspond with the '*all*' and '*selected*' views described above:

- *'all' raw matrix*: Groups the contents of all *comparison matrixes* contained in the *comparison matrix repository*.
- *'selected' raw matrix*: Groups only the contents of the selected *comparison matrix*.

In Figure 13 the two window snapshots of the previous *Graphical User Interface* are identical with the ones shown in Figure 6. This is to illustrate, that each view is the *dotplot* of the correspondent *raw matrix*.

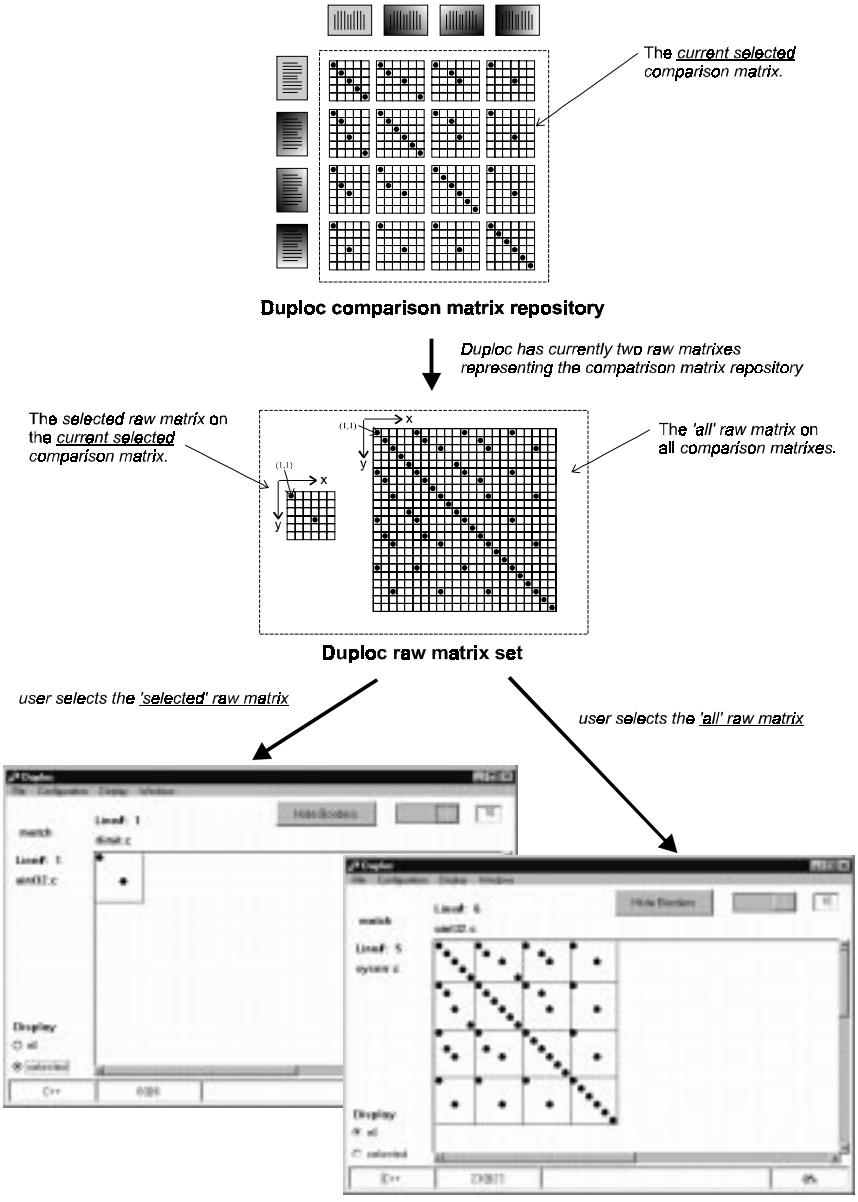


Figure 13: Accessing the *comparison matrix repository* through a set of defined *raw matrixes* stored in the *raw matrix set*.

4.1.2 Representing a large *raw matrix*

The problem of representing a large *raw matrix* is equivalent with the project goal stated above. A large *raw matrix* must be represented by:

- providing an *Information Mural overview binplot*.
- providing navigation means, in order to explore sections of the *raw matrix* with a conventional *dotplot*.

The new implemented *Graphical User Interface* represents a selected *raw matrix* depending on its size with two techniques: A *two level view representation* and a *three level view representation.*

A two level view representation

Up to a certain *raw matrix* size the *Graphical User Interface* offers two views onto the current selected *raw matrix* – see Figure 14:

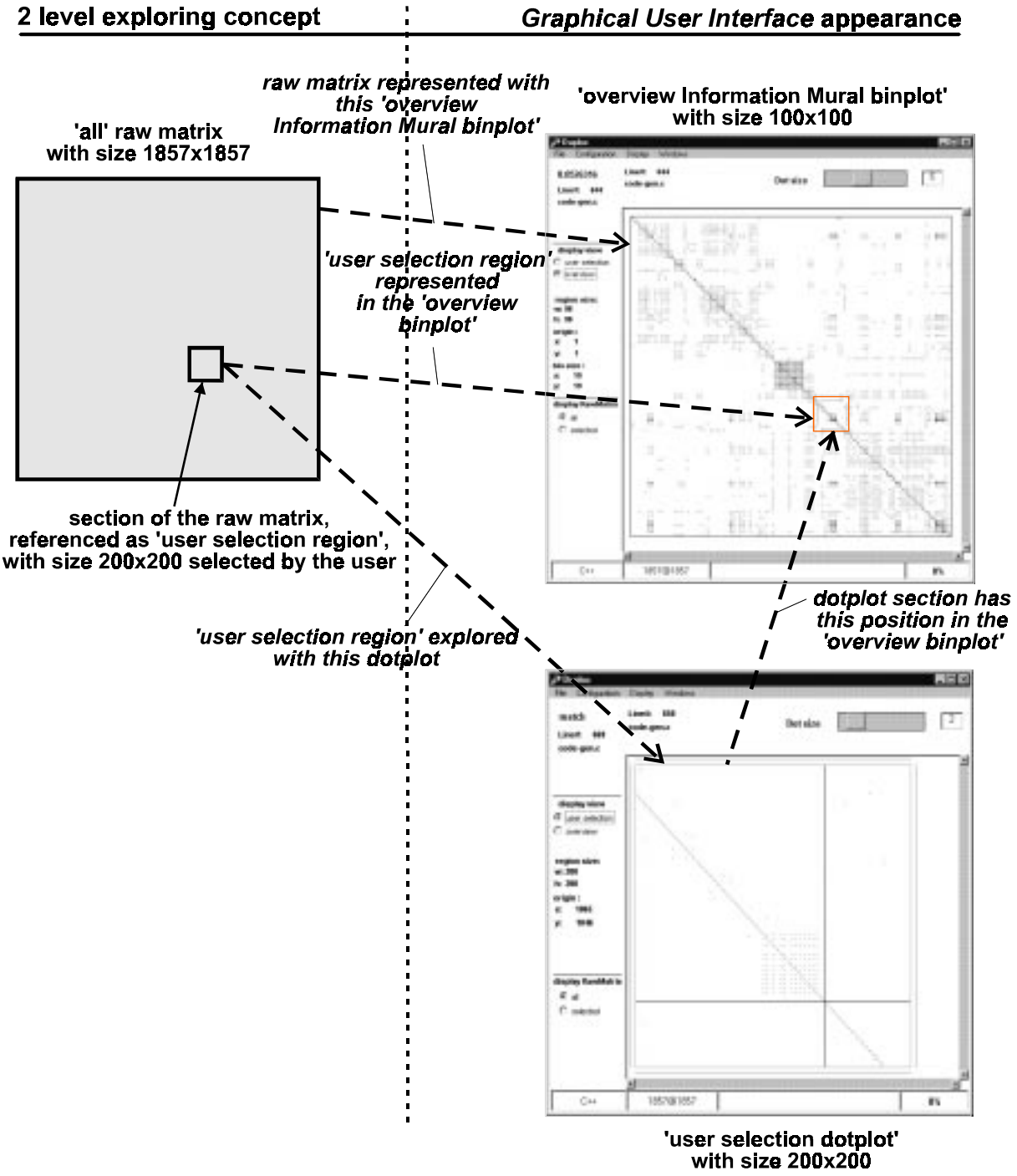


Figure 14: The *two level view representation*.

- *user selection view*: A 200x200 element section of the *raw matrix*, referenced as the *user selection region*, is displayed as a conventional *dotplot*, referenced as the *user selection dotplot* – see bottom right window. Depending on the selected dot size, this *dotplot* varies between 200x200 and 2000x2000 screen pixels. The *user selection dotplot* shown in Figure 14 uses 400x400 screen pixels, because the selected dot size is 2. The *user selection region* has an origin position (top left corner) on the *raw matrix*. The user can explore sections of the *raw matrix* through this *dotplot* by modifying the *user selection origin position* – see next section.
- *overview view*: The complete *raw matrix* is represented as a density *Information Mural matrix* of 100x100 elements. The graphical representation is an *Information Mural binplot* of 100x100 *bin* dots. This *binplot* is referenced as the *overview binplot* – see top right window. Inside this *binplot* an orange rectangle indicates the current position of the *user selection region*. Depending on the selected dot size for drawing the *bin* dots, this *binplot* varies between 100x100 and 1000x1000 screen pixels. The *overview binplot* shown in Figure 14 uses 500x500 screen pixels, because the selected dot size is 5. The size of the *bin regions* depends on the *raw matrix* size: The *raw matrix* is partitioned in 100x100 regions; therefore the width respectively the height of each *bin region* is the 1/100 of the *raw matrix* width respectively height – consult chapter 4, the design chapter, for further explanations about the calculations. Here each *bin* represents a region of 19x19 elements in the *raw matrix*.

The 'displaying barrier' of the two level view representation

In the *overview binplot* the *user selection region* must be represented as an orange rectangle with the right proportions. Above, it is mentioned, that the *user selection dotplot* has a size of 200x200 dots and that the *overview binplot* has a size of 100x100 *bin* dots. Let us consider one moment for both plots a higher limit: e.g. 400x400 dots. With this reasonable size of 400x400 dots for the *overview binplot* and the *user selection dotplot*, a *bin* dot in the *overview binplot* represents max. 400x400 dots in the *user selection dotplot*. Therefore a *raw matrix* with more than 400x400 = 160'000 lines per side can not be represented with this two level strategy; in the correspondent *overview binplot* the *user selection region* would be represented as an orange rectangle around a fraction of a single *bin* dot. With a 200x200 dots limit, this 'displaying barrier' would occur already above a *raw matrix* size of 200x200 = 40'000 lines per side.

A three level view representation

The solution to this 'displaying barrier' problem is to introduce a third level – see Figure 15:

- *user selection view*: This corresponds with the *user selection view* in the *two level view representation* – see bottom left window.
- *overview view*: This corresponds with the *overview view* in the *two level view representation* with a modification: Only a section of the *raw matrix* is represented by this *Information Mural binplot* of 100x100 *bin* dots. This *binplot* is again referenced as the *overview binplot* – see bottom right window. As in the *two level view representation* an orange rectangle indicates the current position of the *user selection region*, if the *overview region* covers it – see rectangle in the bottom right window. As for the *user selection region* the *overview region* has an origin point (top left region corner) in the *raw matrix*. The user can explore the *raw matrix* by modifying this origin position – see next section.
- *super overview view*: This view corresponds with the *overview view* in the *two level view representation* – see top right window. The complete *raw matrix* is represented as a density *Information Mural matrix* of 100x100 elements. The graphical representation is again an *Information Mural binplot* of 100x100 bin dots. This *binplot* is referenced as the *super overview binplot* – see top right window. Inside this *binplot* the orange rectangle indicates the current position of the *user selection region* and the blue rectangle indicates the current position of the *overview region*.

For understanding the criteria, when a switch between *two level view representation* and *three level view representation* occurs, consult the calculations explained in the design chapter – see chapter 5.
With the current settings of 200x200 dots for the *user selection dotplot* and 100x100 *bin* dots for the *super overview* and *overview binplot*, the limit of this current *three level view representation* is a *raw matrix* with a size of 2'000'000 (=100x100x200) lines per side. By choosing higher *binplot* sizes this limit can be pushed further: e.g. 200x200 dots for the *user selection dotplot*, 200x200 *bin* dots for the *super overview binplot* and *overview binplot*; the limit of this *three level view representation* is a *raw matrix* of 8'000'000 lines per side.

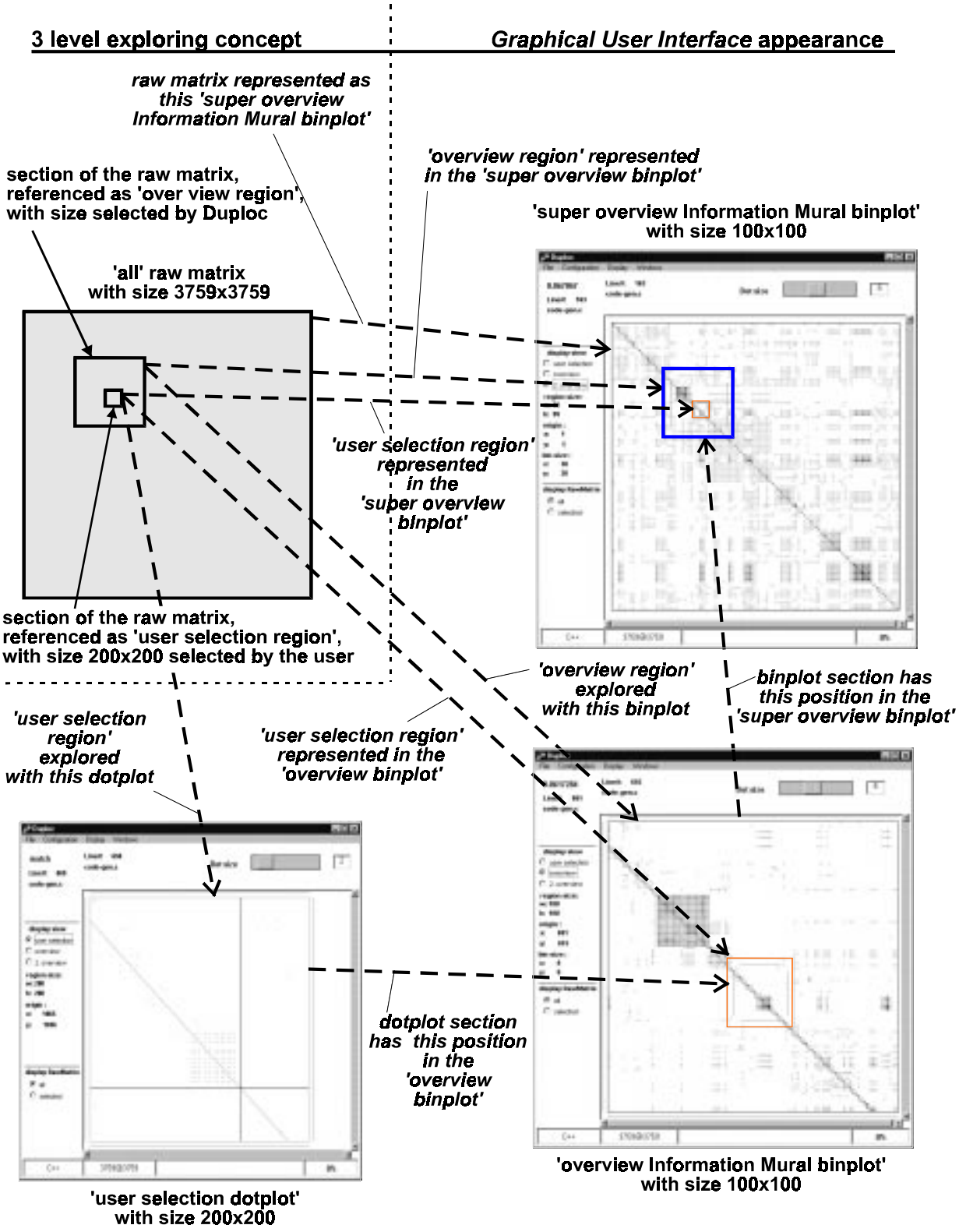


Figure 15: The *three level view representation*.

4.2 Using Duploc in the new Information Mural Interactive Mode

4.2.1 Starting the new version

Consult the '*Duploc Tutorial*' document [2] for a proper installation of the tool. The new *Duploc* version is started by evaluating

```
DuplocInformationMural open
```

in a VISUALWORKS workspace or by selecting the `windowSpec` resource of `DuplocInformationMural` in a Resource Finder and pressing START.

4.2.2 Unchanged loading features from the previous version

The following loading features are invoked by the user like in the previous version[7]:

- Selecting the Source Code Language
- Reading Source Code

4.2.3 Selecting the *raw matrix*

The *raw matrix* selection occurs with the radio button group on the lower left side of the main window labelled with 'display RawMatrix' – see Figure 16.:

- all: Displays the *all raw matrix*.
- selected: Displays the *selected raw matrix*.

4.2.4 Exploring the *raw matrix*

Depending on the *raw matrix* size, which is indicated in the second field of the status bar on the bottom of the *Duploc* window (see Figure 16), a *two level* or *three level view representation* is automatically chosen by *Duploc*. (Consult the design chapter, chapter 5)

Let's begin with a *raw matrix*, which has a size of 1983x1983 elements and which is explored in a *two level view representation* – see Figure 16.

User selection (view) display mode[8]

Each new selected *raw matrix* starts in this display mode. This is indicated in the radio button group **display view**, where **user selection** is set. The **user selection** label and the diagram border are both coloured in orange – the colour attributed to this display mode. The absolute co-ordinate of the upper left corner of the *user selection region* is shown on the left side of the main window under the label **origin:** (N.B. the top left position of the *raw matrix* has co-ordinates (1,1)).The size of the *user selection region* is indicated under the label **region size:** . Red borders around the diagram border indicate that the outer border of the *raw matrix* has been reached, green borders indicate that the *user selection region* can be displaced/moved in that direction.

Common features with the previous version

The following features available in the previous version are also available in this display mode[9]:

- setting the dot size slider
- hiding the delimiting lines between the neighbouring *comparison matrixes.*
- zooming
- examining the source code

Selecting the *user selection region* size

The *user selection region* size can be selected over the following mouse menu items – see Figure 17:
size 200x200, size 400x400, size 600x600 and size 800x800.

[7] see section *2.3 Using Duploc in the original Interactive Mode*
[8] the *user selection view display mode* is referenced as the *user selection display mode* from here on.
[9] see section *2.3.5 Exploring the selected comparison matrix(es)*

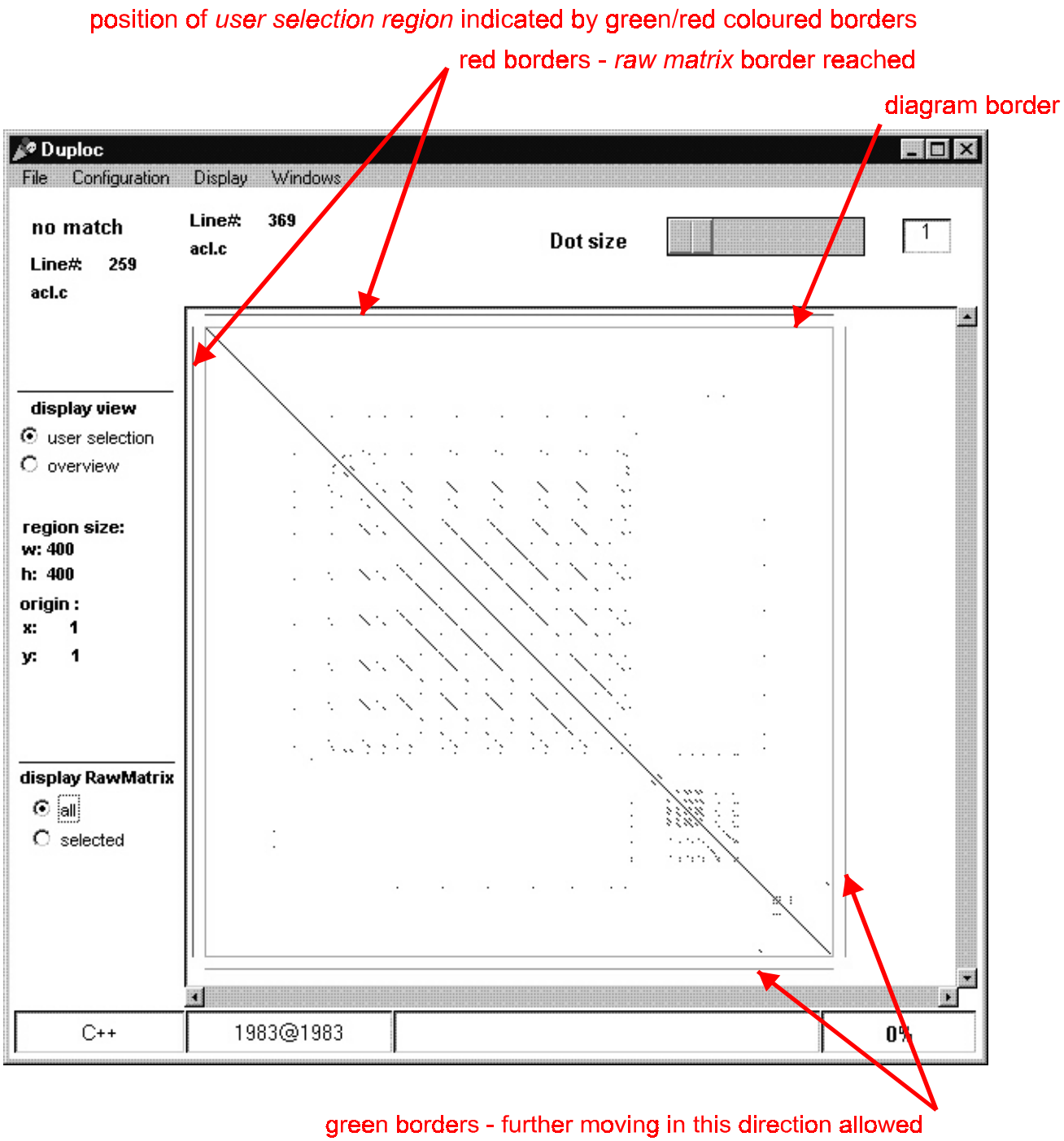


Figure 16: The *'all' raw matrix* in *user selection display mode*.

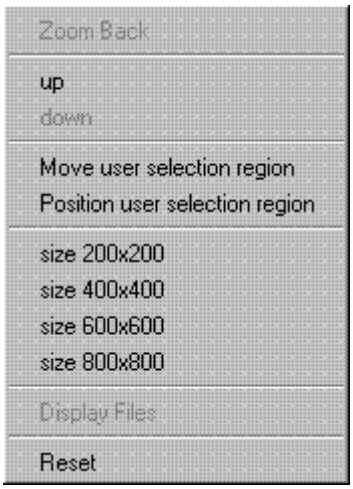


Figure 17: Opened mouse menu in *user selection display mode*.

Modify/Position the *user selection region*
The *user selection region* can be moved inside the boundaries of the *raw matrix* by selecting the Move user selection region mode in the mouse menu (see Figure 17): In the situation of Figure 16, for exploring the *raw matrix* area further to the right, move the cursor (which turns into a hand icon over the *dotplot*) near the right *dotplot* border, click the left mouse button and drag the mouse cursor towards the left *dotplot* border. The *user selection region* can also be positioned by selecting the Position user selection region mode: The *user selection region* centre will be aligned with the clicked position.– this mode results in a sort of two-dimensional scrolling. To De-select either of the two modes, click on Stop Moving/Positioning in the mouse menu.

Switching view level
Selecting the menu item up in the mouse menu corresponds with the selection of the *overview display mode* in the radio button group on the left hand side of the window.

Resetting
By selecting the mouse menu item Reset the *user selection region* is resized to 400x400 elements and repositioned at (1,1).

Overview (view) display mode[10] in a *two level view representation*

In the *two level view representation*, the complete *'all' raw matrix* introduced in Figure 16 is represented with this *overview Information Mural binplot* – see Figure 18. The radio button group **display view** indicates **overview**. This **overview** label and the diagram border are both coloured in blue – the colour attributed to this display mode. The *overview Information Mural binplot* represents the complete *'all' raw matrix*. Therefore the absolute co-ordinate of the upper left corner is set at (1,1) – see under the label **origin:**. The size of the *overview region* is computed by multiplying the number of dots in the *Information Mural binplot* indicated under the label **region size:**[11] and the *bin* size indicated under the label ***bin* size:**. In Figure 18 the represented *overview region* has 100x20 = 2000 elements per side – this is a larger region as the *raw matrix region* with 1983 elements per side. The current *user selection region* is represented by an orange rectangle[12] in the *binplot*. As in the *user selection display mode* the same colour coding with the red/green borders is applied[13], but here the red/green lines are drawn inside the orange rectangle.

Common features with the *user selection display mode*
The following features available in the *user selection display mode* are also available in this display mode:
- setting the dot size slider
- zooming
- resetting

Mouse pointer information
The *bin* density under the current mouse cursor position is indicated in the upper left window corner.

Move/Position the *user selection region*
The *user selection region* can be moved inside the boundaries of the *raw matrix* by selecting the Move user selection region mode in the mouse menu (see Figure 19): In the situation of Figure 18, for moving the *user selection region* towards the right, move the mouse cursor (which turns into a hand icon over the *binplot*) over the binplot area, click the left mouse button and drag the mouse cursor towards the right *binplot* border. The *user selection region* can also be positioned by selecting the Position user selection region mode: The *user selection region* centre will be aligned with the clicked position. To De-select either of the two modes, click on Stop Moving/Positioning in the mouse menu.

Enable/Disable showing objects
In order to appreciate the *Information Mural binplot*, the drawing of the orange rectangle, which represents the *user selection region*, can be omitted.

Spying *bin*s
By selecting the mouse menu item Spy bin region and by clicking on any *bin* dot, the contents of the correspondent *bin region* is displayed next to the *bin* dot in a red painted rectangle – see Figure 20.

Switching view level
Selecting the menu item down in the mouse menu corresponds with the selection of the *user selection display mode* in the radio button group on the left hand side of the window.

[10] the *overview view display mode* is referenced as the *overview display mode* from here on.
[11] the label **region size:** should be renamed **plot size:**.
[12] remember, that the colour orange was attributed to the *user selection display mode*.
[13] see section *user selection display mode* on page 19

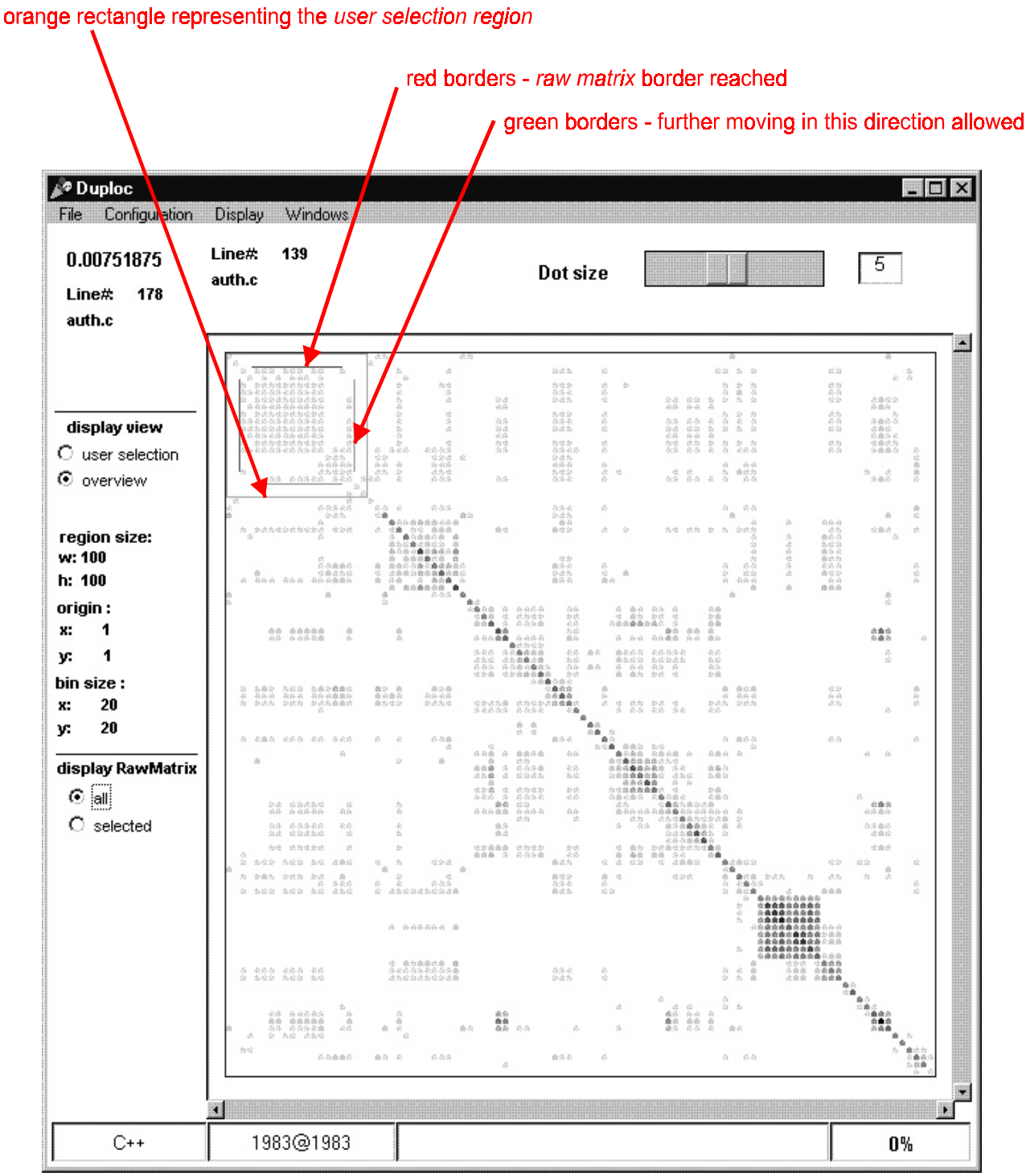


Figure 18: *'all' raw matrix* in *overview display mode* in a *two level view representation*.

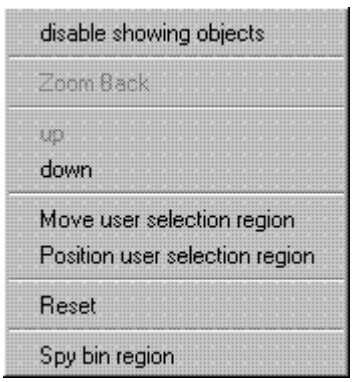


Figure 19: Opened mouse menu in *overview display mode* in a *two level view representation*.

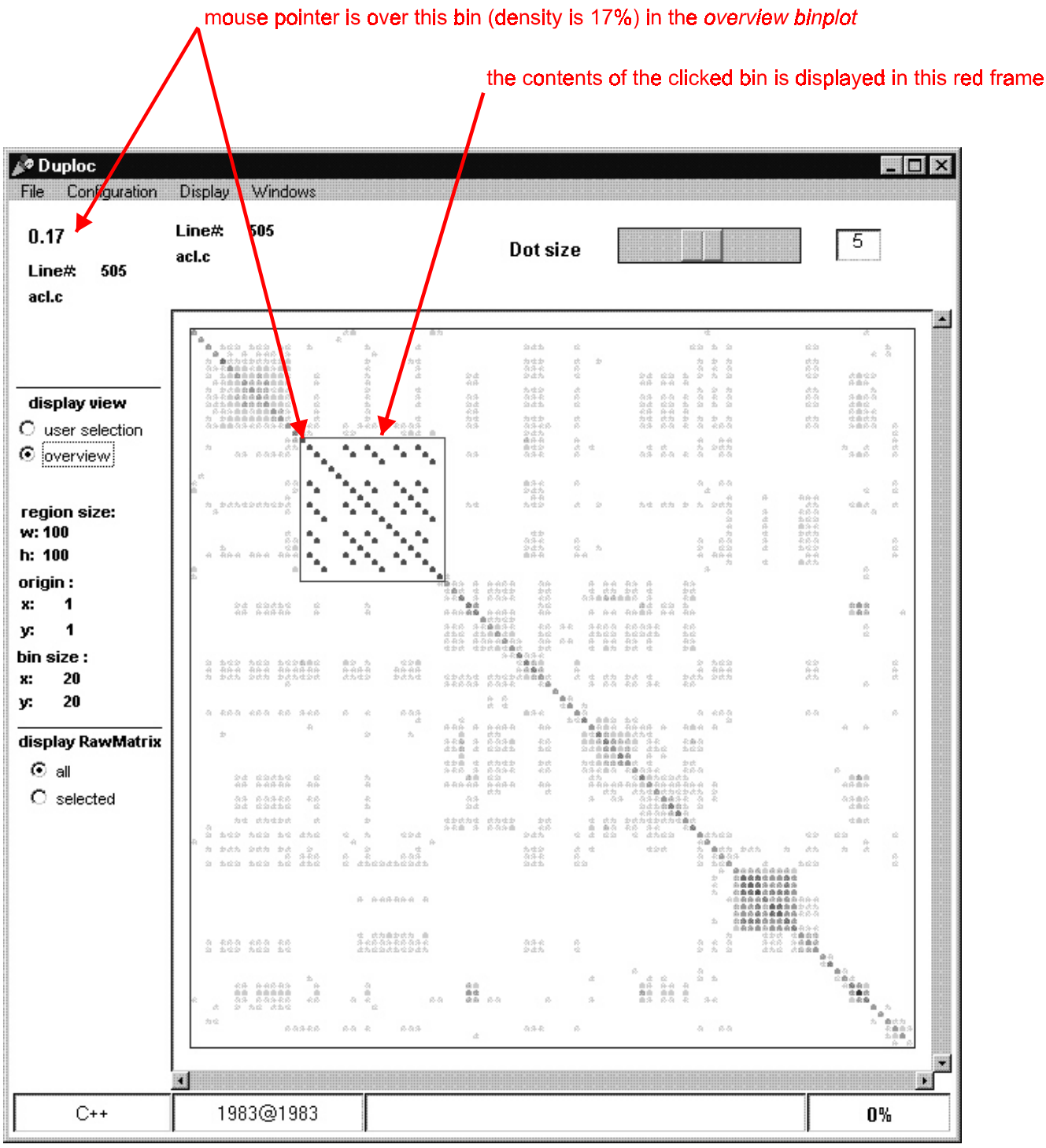


Figure 20: Spying a *bin* in the *overview display mode* in a *two level view representation*.

Let's continue with a *raw matrix*, which has a size of 10703x10703 elements and which is explored in a *three level view representation* – see Figure 21.

Super overview (view) display mode[14] in a *three level view representation*

In the *three level view representation*, the complete *'all' raw matrix* is represented with this *super overview Information Mural binplot* – see Figure 21. The radio button group **display view** indicates **2. overview** – this label text was chosen because of the restricted space. This **2. overview** label and the diagram border are both coloured in black – the colour attributed to this display mode. The *super overview Information Mural binplot* represents the complete *'all' raw matrix*. Therefore the absolute co-ordinate of the upper left corner is set at (1,1) – see under the label **origin:**. The size of the *super overview region* is computed by multiplying the number of *bin* dots in the *Information Mural binplot* indicated under the label **region size:** and the *bin* size indicated under the label **bin size:**. In Figure 21 the represented *super overview region* has 100x108 = 10800 elements per side. The current *user selection region* is represented by an orange rectangle and the current *overview region* is represented by a blue rectangle in the *binplot*. The *overview region* can be moved or positioned in this display mode. The red respectively green lines on the inside of the blue rectangle indicate if a side did respectively did not reach the outer border of the *raw matrix*. The *user selection region* can not be moved in this display mode, because the moved distance would correspond with a multiple of the *bin region* size. If this *bin region* size is much bigger then the actual *user selection region*, then you would loose the precision in positioning it. Therefore the inside of the orange rectangle on the *super overview binplot* is drawn with red lines on four sides. Moving or positioning the *user selection region* has to occur in the *overview display mode* – see further down.

Common features with the *overview display mode* in the *two level view representation*

The following features available in the *overview display mode* are also available in this display mode:

- setting the dot size slider
- zooming
- resetting
- mouse pointer information
- enable/disable showing objects
- spying bins

Move/Position the *overview region*

The *overview region* can be moved inside the boundaries of the *raw matrix* by selecting the Move overview region mode in the mouse menu (see Figure 22): In the situation of Figure 21, for moving the *overview region* towards the right, move the mouse cursor (which turns into a hand icon over the *binplot*) over the binplot area, click the left mouse button and drag the mouse cursor towards the right *binplot* border. The *overview region* can also be positioned by selecting the Position overview region mode: The *overview region* centre will be aligned with the clicked position. To De-select either of the two modes, click on Stop Moving/Positioning in the mouse menu.

Switching view level

Selecting the menu item down in the mouse menu corresponds with the selection of the *overview display mode* in the radio button group on the left hand side of the window.

Overview (view) display mode in a *three level view representation*

Figure 23 shows the *overview display mode* for the *raw matrix* introduced in Figure 22. This display mode corresponds with the *overview display mode* for a *two level view representation* with some modifications discussed above: Because this *overview region* covers only a part of the *raw matrix* (in the example of Figure 23 it is 1500x1500 elements), the absolute co-ordinate of the upper left corner is displayed under the label **origin:**. If the current *user selection region* is covered by this *overview region*, then it is represented by an orange rectangle.

Common features with the *overview display mode* in a *two level view representation*

This display mode supports all features presented in the *overview display mode* in a *three level view representation*:

Move/Position the *overview region*

The Move overview region mode and Position overview region mode allow to move and position the *overview region* in the same way as the two equivalent modes in the *user selection display mode* allow to do with the *user selection region* (see Figure 24). The menu item Focus on user selection aligns the *overview region centre* with the current *user selection region* centre.

Switching view level

Selecting the menu item down/up in the mouse menu corresponds with the selection of the *user selection* / *super overview display mode* in the radio button group on the left hand side of the window.

[14] the *super overview view display mode* is referenced as the *super overview display mode* from here on.

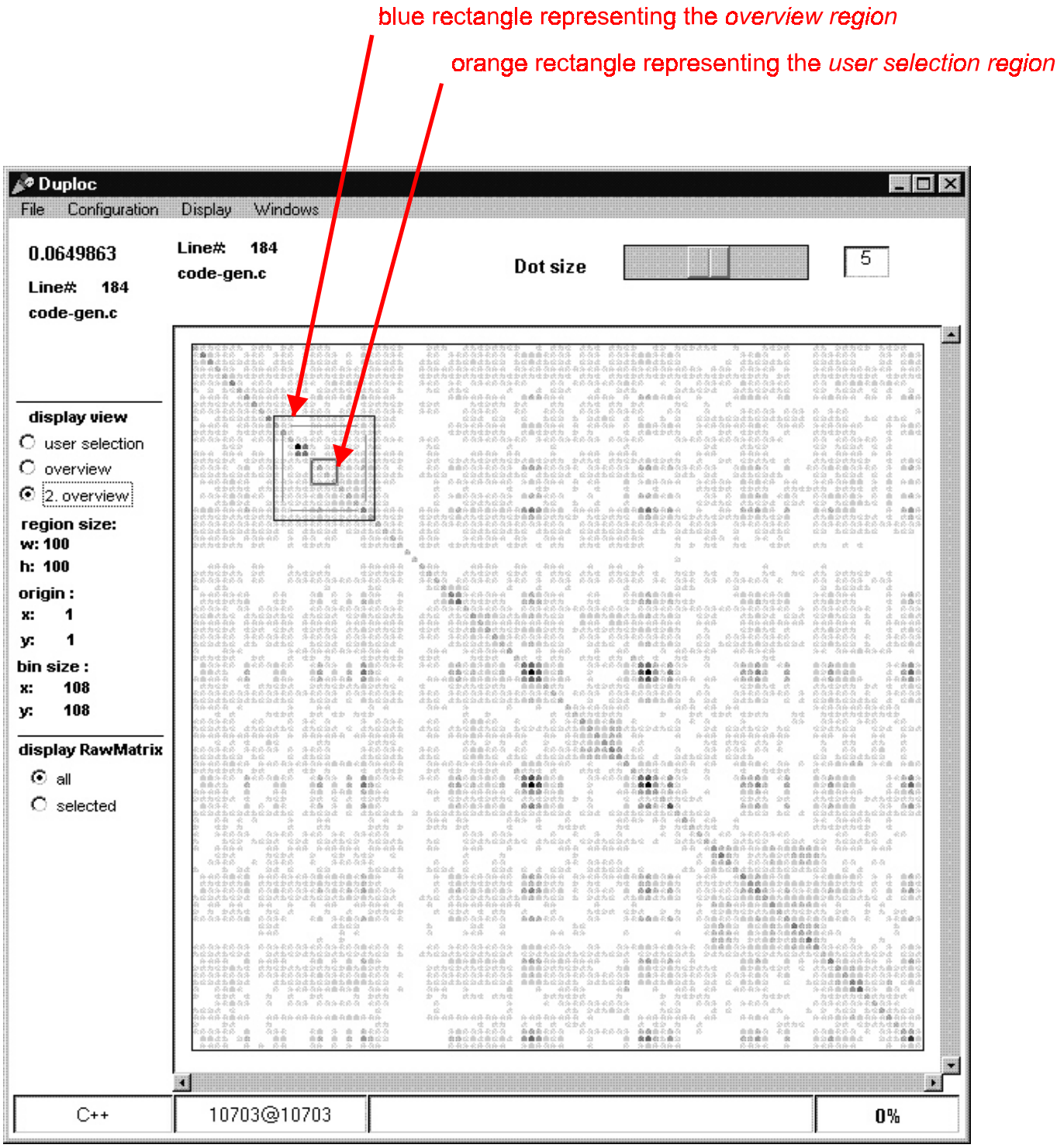


Figure 21: *'all' raw matrix* in *super overview display mode* in a *three level view representation*.

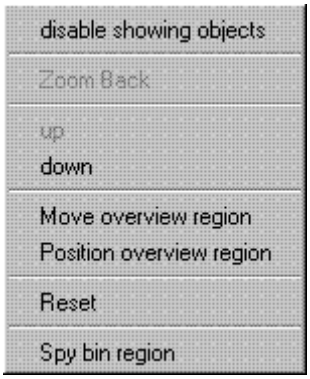


Figure 22: Opened mouse menu in *super overview display mode* in a *three level view representation*.

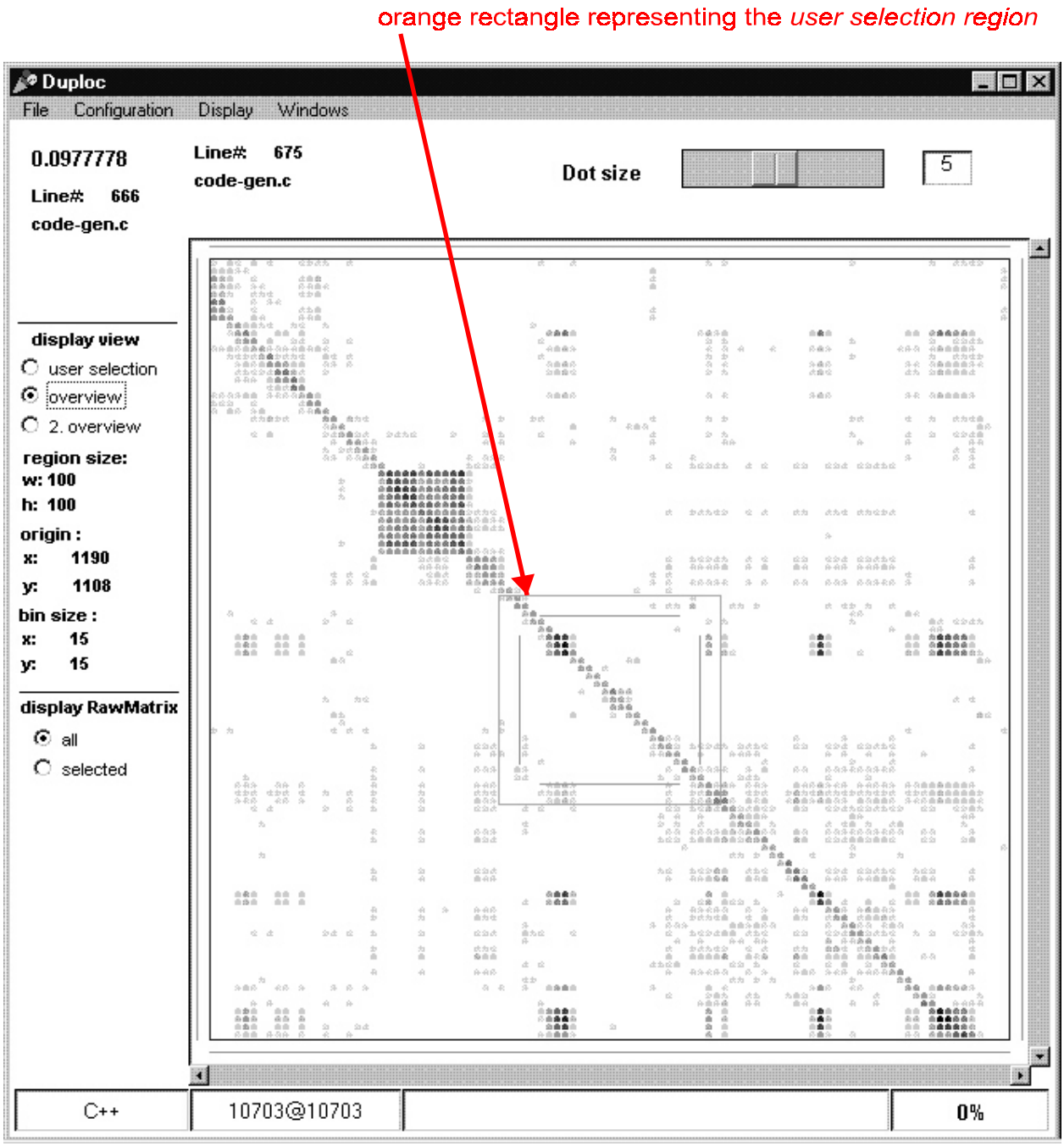


Figure 23: *'all' raw matrix* in *overview display mode* in a *three level view representation*.

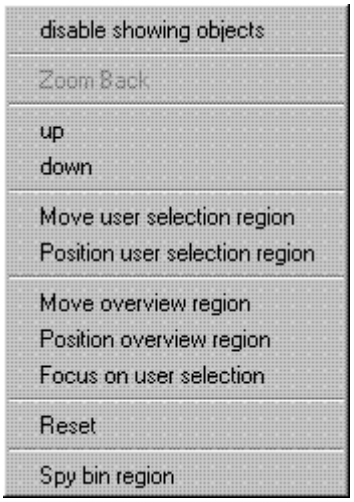


Figure 24: Opened mouse menu in *overview display mode* in a *three level view representation*.

4.2.5 The *bin* value colouring function

A *binplot* is the graphical representation of a density *Information Mural matrix*. The *bin* dot shading occurs with a defined *bin value colouring function* (see Figure 12). A *bin*-value, in the range from 0.0 to 1.0 is mapped to a grey level between white and black. By modifying the function, different *binplots* can be obtained, which can help to identify some interesting regions of the *raw matrix*. *Duploc* binds to the *super overview* and the *overview Information Mural binplot* two independent *bin value colouring functions*. The *bin value colouring function* settings of the current *Information Mural binplot* visible in the current display mode are presented in an auxiliary window – the *bin colouring tool*. This tool can be started in the *super overview* and *overview display mode* from the mouse menu under open <*bin* value colorer> tool, if it is not open yet – see Figure 25b. Figure 26 shows the starting appearance of the tool, which displays the settings of the *bin value colouring function* used for generating the *super overview Information Mural binplot* displayed in Figure 25.

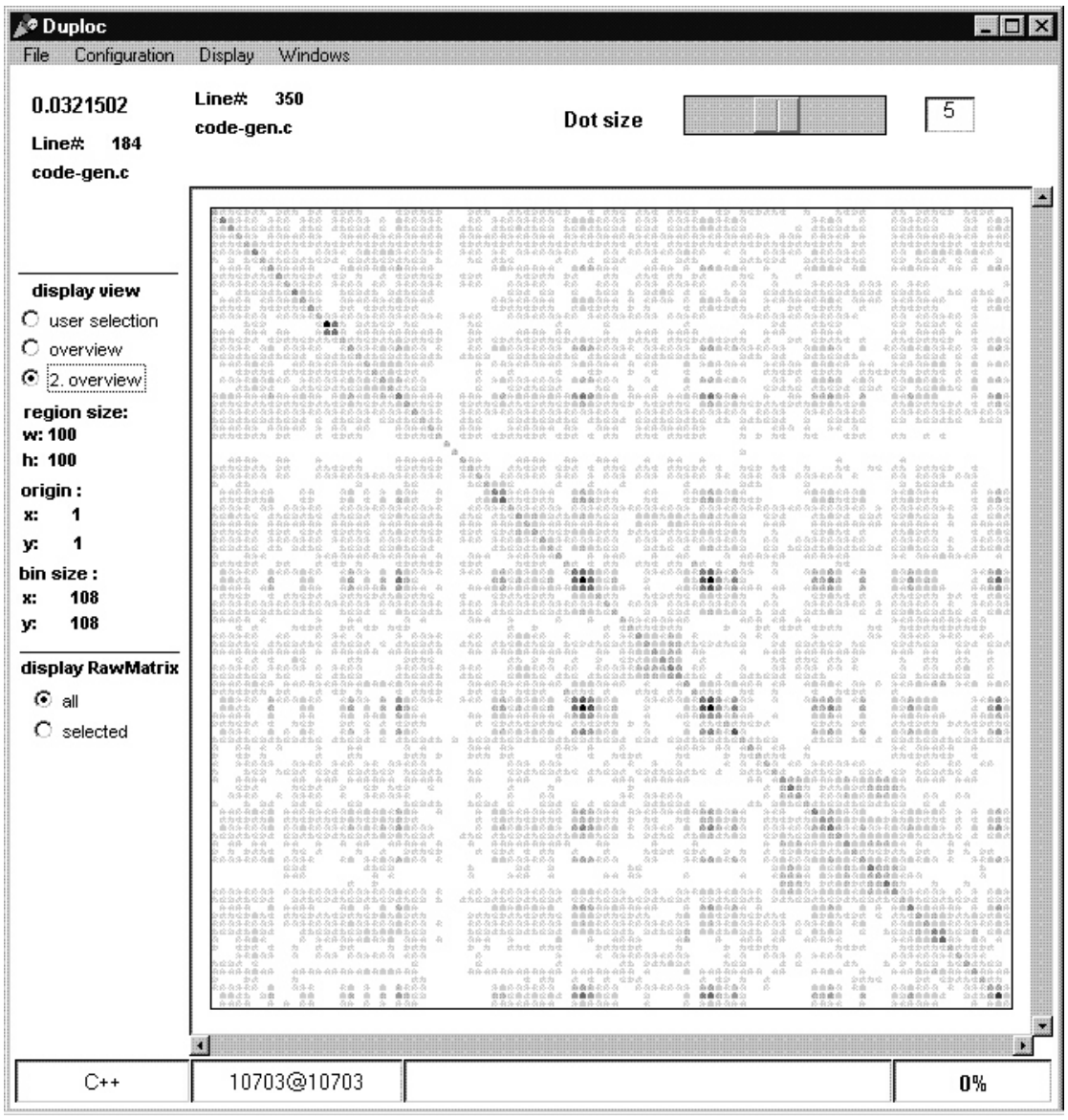


Figure 25: 'Super Overview' Information Mural binplot.

Figure 25b: Opened mouse menu in *super overview display mode* - '*open <bin value colourer> tool*' mouse menu item visible.

The *available input range* is the range from the smallest *bin* value (which is bigger than zero) to the largest *bin* value that occur in the bound *Information Mural matrix*. The *available input range* of this *super overview Information Mural matrix* displayed in Figure 26 has a minimum of 8.57339e-5 and a maximum of 0.066358. Each *bin value colouring function* defines an *input range*, which is equal or smaller then the actual *available input range*. By clicking on the button Available Range, the *input range* is set on the *available input range*. The *information box* on the bottom right part of the window in Figure 26 displays always the current set *input range*.

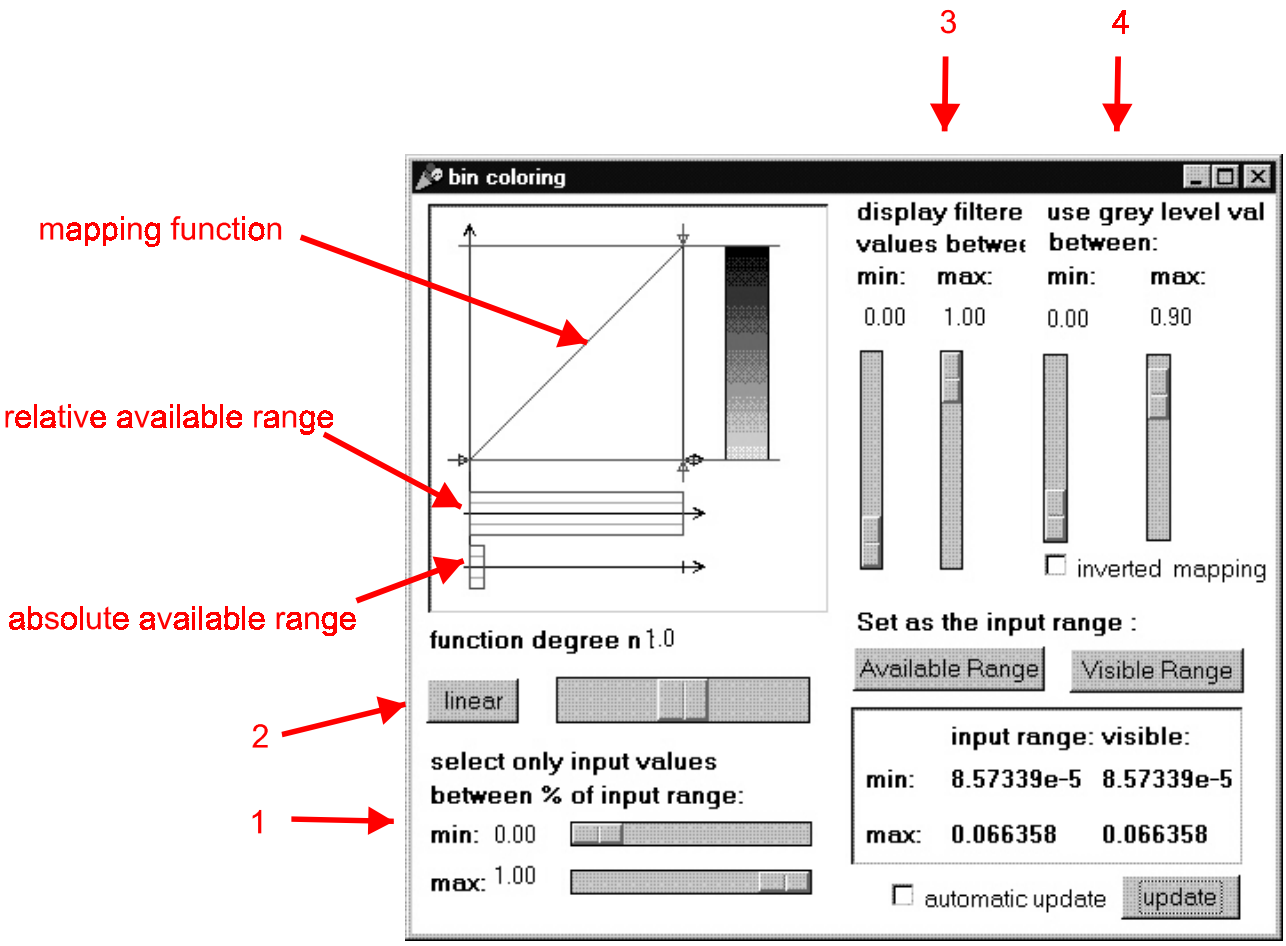


Figure 26: bin coloring tool - starting appearance.

The *available input range* is not displayed numerically; only a graphical representation is available. Three horizontal axis below the mapping function show the relationship between the maximal range (from 0.0 to 1.0), the *available input range* and the *input range*. All three axis have a vertical mark, so that the length between the left vertical axis and this mark before the arrow head represents one horizontal unit. On the middle and bottom axis two types of coloured rectangle appear: A rectangle coloured in orange with the smaller height, referenced as the *available input range rectangle*, represents the *available input range* relatively to the horizontal unit. A rectangle coloured in red with the higher height, referenced as the *input range rectangle*, represents the *input range* relatively to the horizontal unit. (Figure 29 shows this different rectangle heights better then Figure 26.) The bottom horizontal axis represents the maximal range (from 0.0 to 1.0). Therefore the position and width of the *available input range rectangle* represents the *available input range* in reference to the maximal range and the position and width of the *input range rectangle* represents the *input range* in reference to the maximal range. The middle horizontal axis represents the *available input range*. Therefore the *available input range rectangle* covers the middle horizontal axis. The position and width of the *input range rectangle* represents the *input range* in reference to the *available input range*.
The upper horizontal axis represents the current *input range*. The mapping function, which is displayed in red, maps a percentage of the *input range* to the vertical grey scale. This percentage can be set with the two sliders indicated by arrow #1. The function degree can be set with the slider indicated by arrow #2. Currently the function degree in Figure 26 is one; therefore a linear mapping is selected. The two sliders indicated by arrow #3 delimit the function output range, which is effectively mapped linearly onto the grey level range selected with the two sliders indicated by arrow #4.
Figure 27 shows some selected linear restrictions on the current *input range* (which in this example still corresponds with the *available input range*):

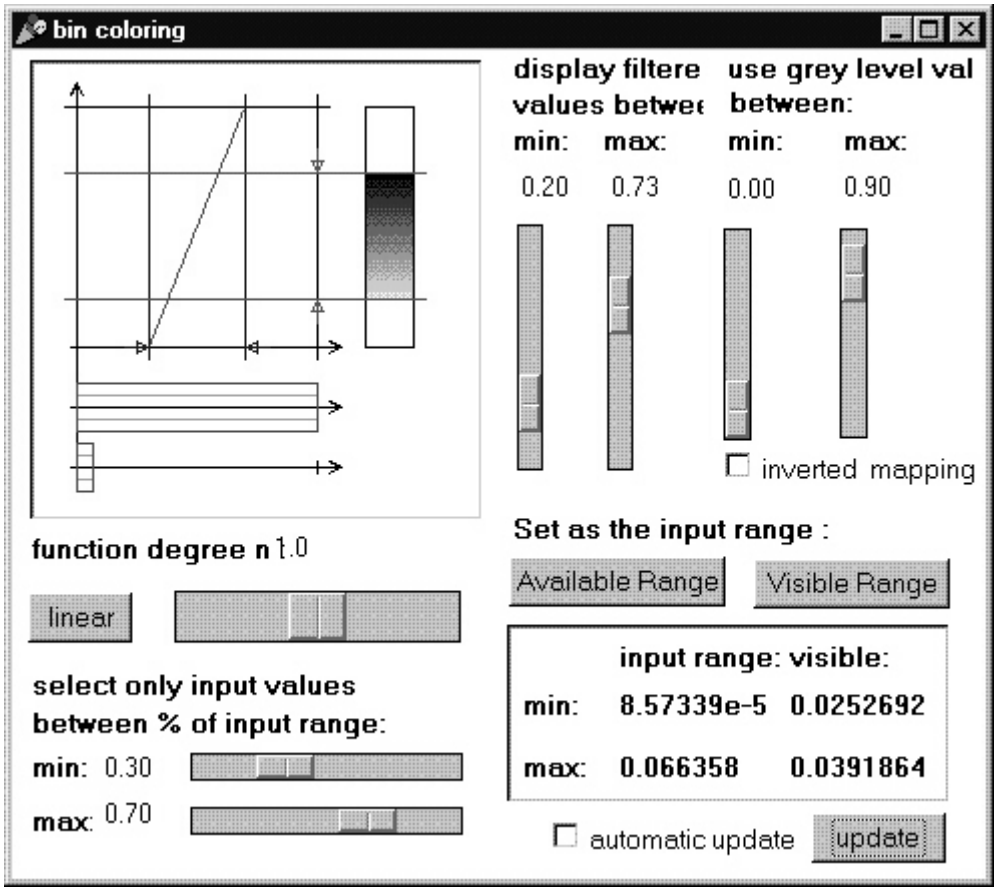


Figure 27: bin coloring tool - linear restrictions selected

The mapping function is only applied between the 30th% and the 70th% of the *input range*. The function degree remains on one for achieving a linear mapping. Only the values lying between the 20th% and the 73th% of the output range are effectively mapped linearly onto the grey level range between 0.0 (white) and 0.9 (dark) – see filled rectangle on the right of the mapping function. Once the update button is pressed the *super overview Information Mural binplot* will appear like in Figure 28. Only *bin*s between 0.0252692 and 0.0391864 are visible – this *visible range* is indicated in the *information box*. By pressing the button Visible Range the *input range* can be restricted on this current *visible range* – see Figure 29. Repeating the updating will only reproduce the *binplot* shown in Figure 28.

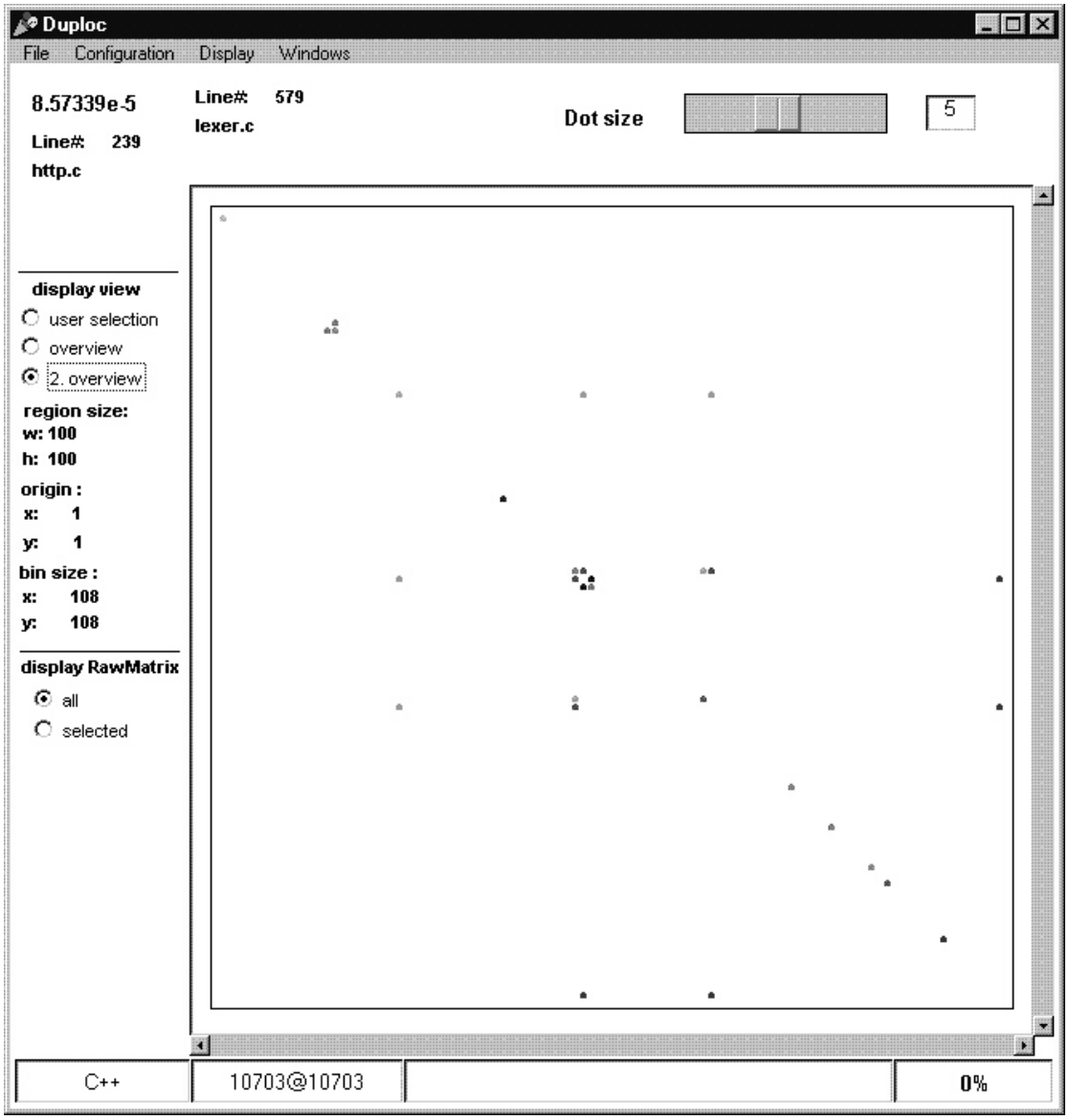


Figure 28: Resulting *super overview Information Mural binplot* from the applied linear restrictions.

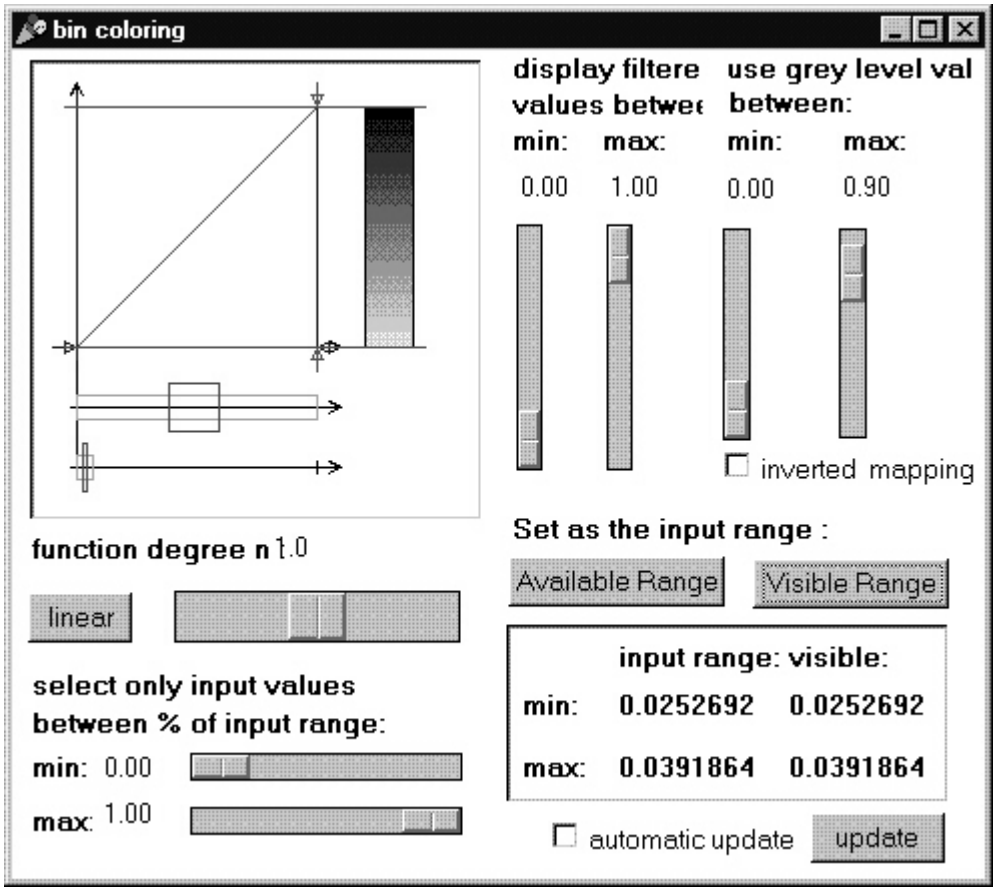


Figure 29: bin coloring tool - visible range set as input

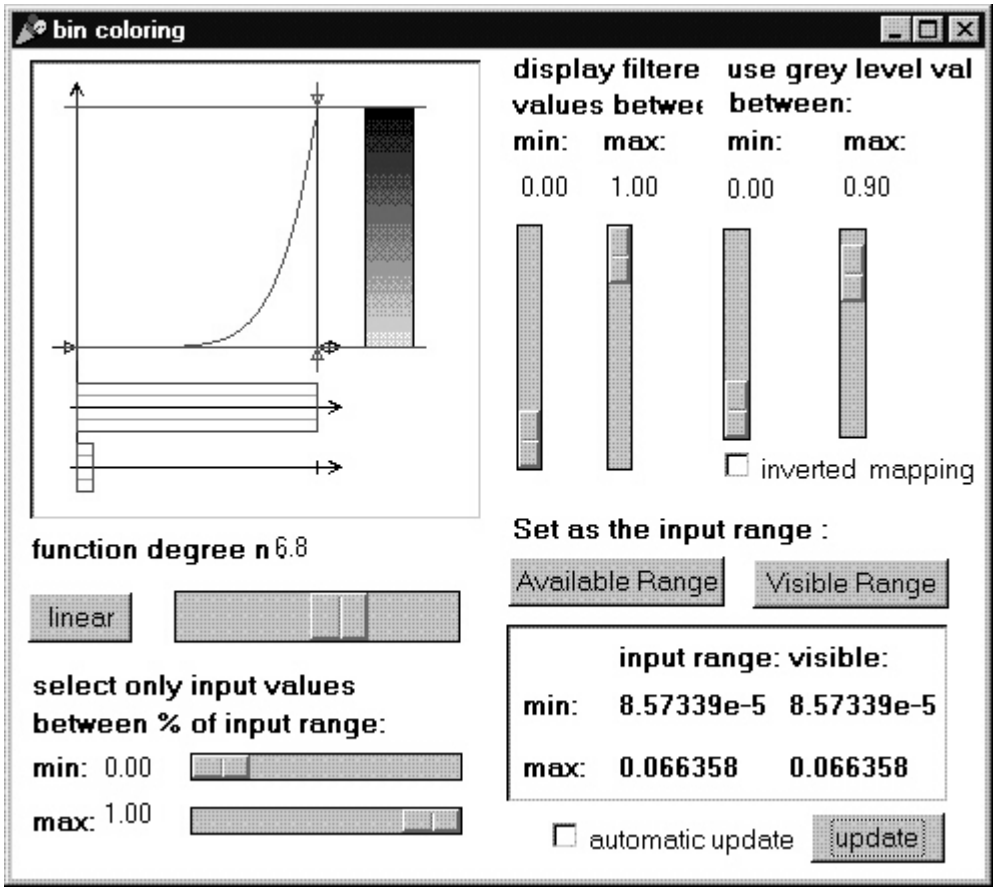


Figure 30: bin coloring tool - available range selected & function degree 6.8 selected.

In order to demonstrate the usefulness of this tool, the *input range* is restored on the *available input range*. By choosing a high function degree of 6.8 like shown in Figure 30 only dense *bins* will appear darker – see Figure 31. With this simple settings the dense region in any current set *input range* will also appear darker.

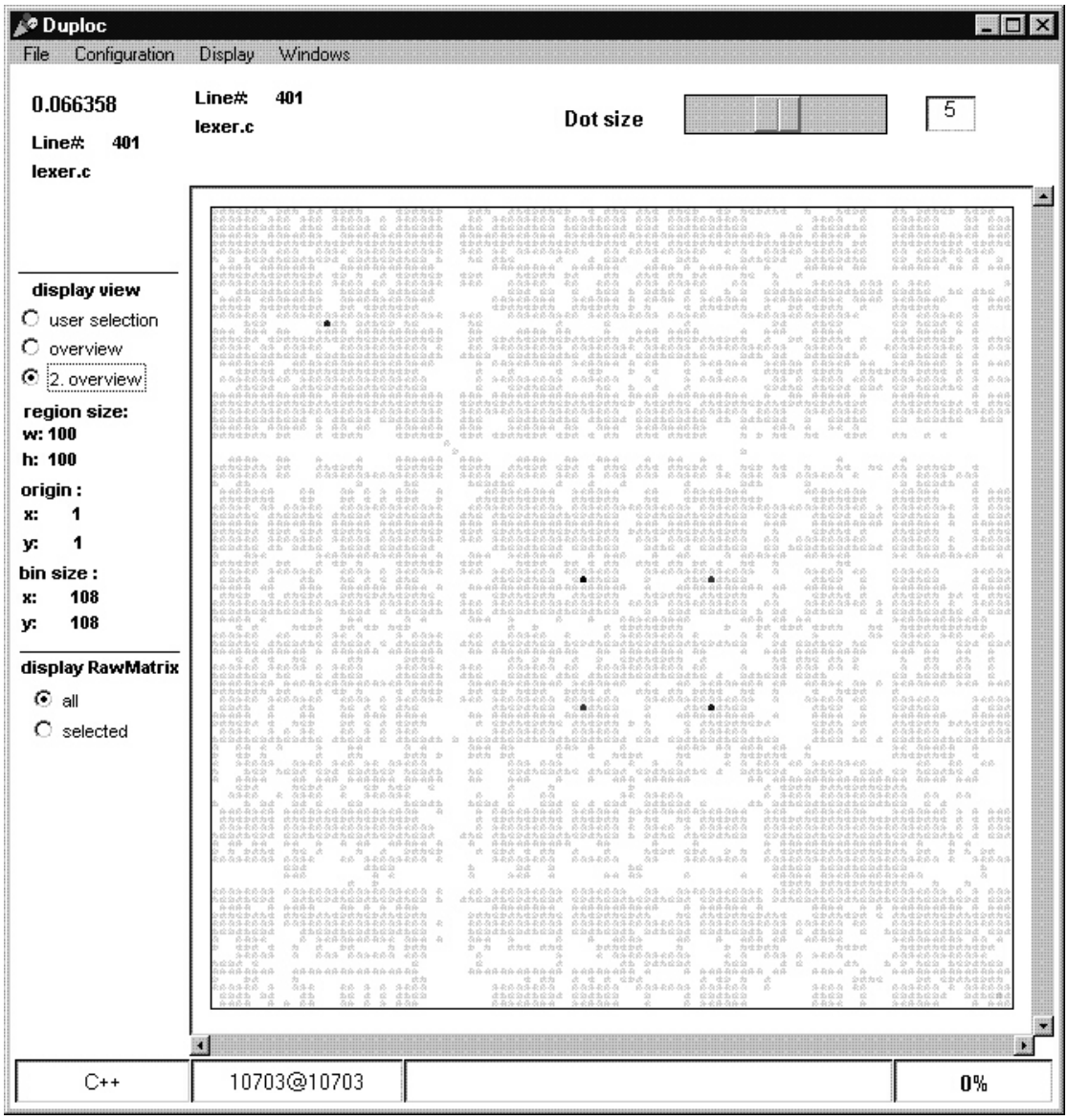


Figure 31: Resulting *super overview Information Mural binplot* from the applied non linear mapping - function degree 6.8

5 Design

5.1 Introduction

The design of the application extension is explained in two sections:

- '*System outline*'
- '*System details*'

The system is implemented in Smalltalk – therefore this chapter makes references to Smalltalk expressions: e.g. 'self'.

System outline
A sequence of diagrams explains the outline of the system. The system is designed according to an implementation concept, which is described in the section *5.3.8 Implementation concepts*. These diagrams are based on a defined graphical notation, which reflects the used implementation concept. This graphical notation is described in section *5.3.9. Used graphical notation*

System details
This section contains next to the mentioned implementation concept and graphical notation the details about the used models, the behaviour of certain classes and update protocols between classes and its dependants.

5.2 System outline

A serie of 15 diagrams are presented in four parts for describing the implemented design:

- part I (Diagram 1) presents the main application structure around two 'clouds' of classes: One cloud, referenced as the *graphical cloud*, groups classes used for the MVC pattern and one cloud, referenced as the *model cloud*, groups classes used for representing the current *raw matrix*.
- part II (Diagram 2a, 2b) presents the class hierarchy, which models the *user selection*, *overview* and *super overview regions* introduced above.
- part III (Diagram 3a – 3h) presents the details of the *graphical cloud*.
- part IV (Diagram 4a – 4d) presents the details of the *model cloud*.

The diagrams are inserted after the following text, which describes them. N.B. all mentioned sections are subsections of the section *5.3 System details.*

Diagram 1. ***Classes around the application model class DuplocInformationMural***

The Diagram 1 shows the application structure around the class *DuplocInformationMural* (highlighted by a shadow), which is a subclass of *DuplocApplication*, the application model of the old implementation. The *SelectionInList* class references the set of defined *raw matrixes*. A *raw matrix* is modelled with the *RawMatrix* class. The *DuplocPresentationModel* instance has a method '**diagramModel**', which returns a reference on the current selected *RawMatrix* instance. The bottom cloud on the diagram is the *model cloud*: It represents the classes used by the *DuplocPresentationModel* instance for representing a current *RawMatrix* instance. Depending on the current *raw matrix* size, a *two level* or *three level view representation* of the current *raw matrix* is realised. The MVC model is formed by the *DuplocPresentationModelView* class, *DuplocPresentationModelController* class and *DuplocPresentationModel* class. The *DuplocPresentationModelView* instance is signalled by the window system to (re)draw the GUI view area and the *DuplocPresentationModelController* instance is signalled to respond to the mouse events - see event symbols. Both classes can assume different states: The *DuplocPresentationModelView* class must display different diagrams and the *DuplocPresentationModelController* class must support different mouse functions. These states are implemented with the 'state' design pattern, where the incoming events are forwarded to the state classes. The top cloud on the diagram is the *graphical cloud*, which groups the 'view' and 'controller' state classes. It also contains a class for displaying via the *UIBuilder* class the *raw matrix* information below the current mouse cursor position. It is important to remember the two different types of lines used for the two clouds - they are used throughout the diagrams as an orientation help. The selection of the current *raw matrix* by the user through the corresponding radio button will send an event to the *SelectionInList* instance.
The selection of the current display mode by the user through the corresponding radio button will send an event to the *ValueHolder* instance, which is accessed by the *DuplocPresentationModelView* instance with the '**viewSelector**' method.

Diagram 1. Classes around the application model class *DuplocInformationMural*

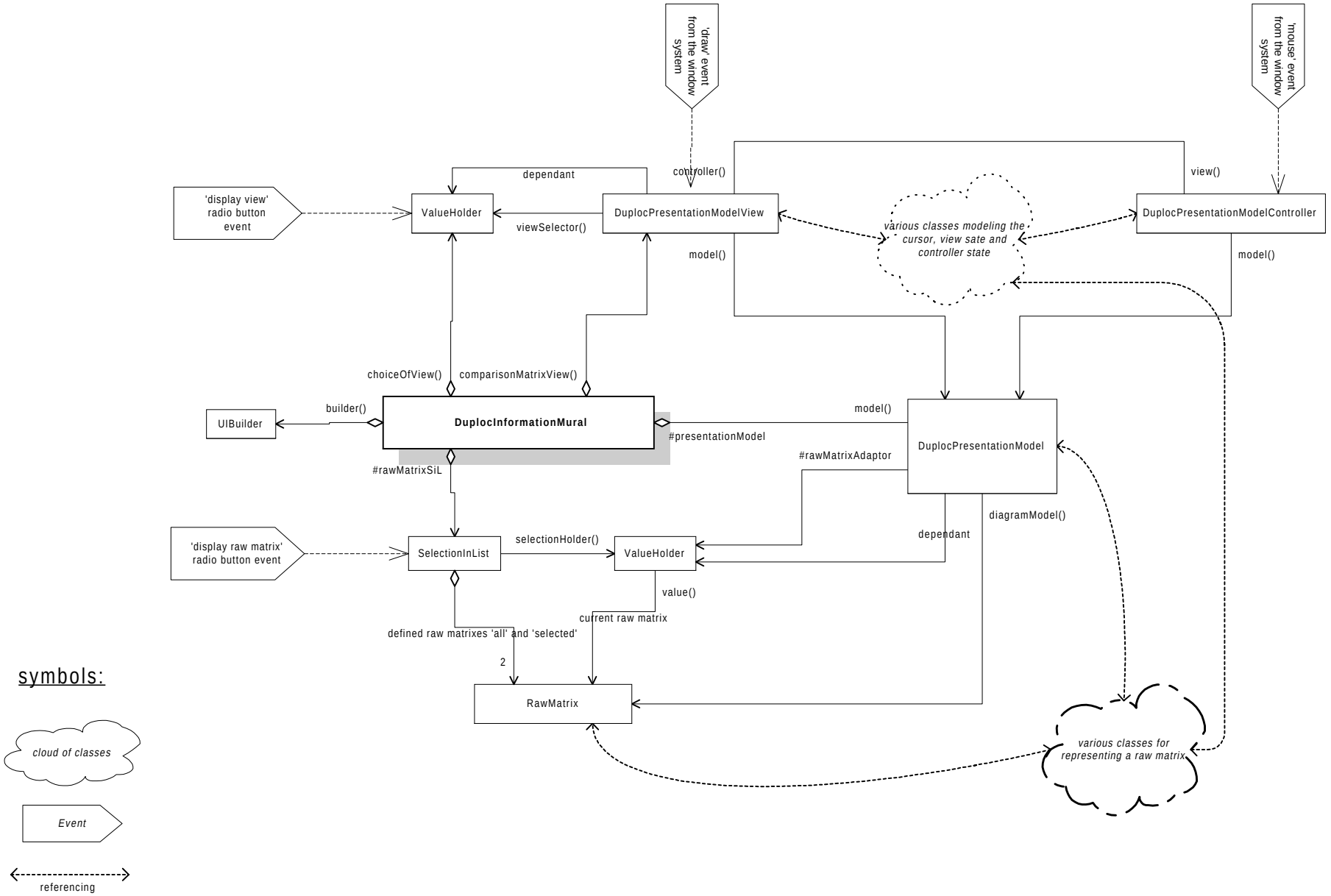


Diagram 2a. *RawMatrix and AbstractRawSubMatrix* class*es*

The *RawMatrix* class represents the *raw matrix* model. (see section *4.1.1 The raw matrix set*)

- ***Features of the RawMatrix class***:
  - It has a height *n* and width *m*.
  - The upper left corner has co-ordinates *(1,1)*.
  - and the bottom right corner has co-ordinates *(n, m).*
- ***Methods summary of the RawMatrix class***:
  - The class provides a set of methods for accessing the comparison results: e.g. '**hasMatchAt:aPoint**' returns a Boolean for indicating the comparison result at the co-ordinates **aPoint** – **true** for a match and **false** for no match.
  - It also provides so called enumerating methods: e.g. '**forSubMatrix:aRectangle do:aBlock**' evaluates for each found match lying in the specified region **aRectangle** the block **aBlock**. There are further enumerating methods '**columnBordersIn:aRectangle do:aBlock**' and '**rowBordersIn:aRectangle do:aBlock**', which return within the bounds of the specified region **aRectangle** all first columns respectively first rows belonging to the next file in the *raw matrix*.

The *user selection*, *overview* and *super overview regions* introduced above are each realised as an own class – this is described below. All three regions are a sort of *raw sub matrix*:

Definition: A ***raw sub matrix*** represents a region of an observed *raw matrix*. It has an origin position inside the *raw matrix* and a certain size.

There is an important consistency condition:

Consistency conditions: A *raw sub matrix* must always represent a sub-region of an observed *raw matrix*. It must be contained in the *raw matrix* area.

The *AbstractRawSubMatrix* class models a *raw sub matrix*:

- ***Features of the AbstractRawSubMatrix* class**:
  - The attribute *region* stores an instance of the *Rectangle* class: This rectangle represents the covered *raw matrix* area. The upper left corner, referenced as the *origin point*, is expressed in the co-ordinate system of the *raw matrix*.
  - The region has a height *n'* and width *m'*.
  - It has a local co-ordinate system in its covered region:
    - The upper left corner has co-ordinates *(0,0)*
    - and the bottom right corner has co-ordinates *(n' - 1, m' - 1).*
- ***Methods summary of the AbstractRawSubMatrix* class**:
  - The class provides a set of methods equivalent to them available in the *RawMatrix* class, which uses the local co-ordinate system.
  - The origin position of the region can be moved to a new position with the method '**movedTo:aPosition**'. A position, which would move the *raw sub matrix region* outside the *raw matrix*, is refused. Instead a linear approximation is made. The 'Consistency condition' is guaranteed.

The *AbstractRawSubMatrix* class is designed using the 'self topology' implementation concept – see section *5.3.8.3 The concept of 'self topology'*. Therefore this class and all its subclasses are independent from the application topology. The reference to the *RawMatrix* instance is returned by sending the message 'self topology rawMatrix'.

Each class, which is a subclass of the *AbstractRawSubMatrix* class, is a dependant of the current *RawMatrix* class. How this dependency is realised is explained in the fourth part.

Diagram 2a. *RawMatrix* and *AbstractRawSubMatrix* classes

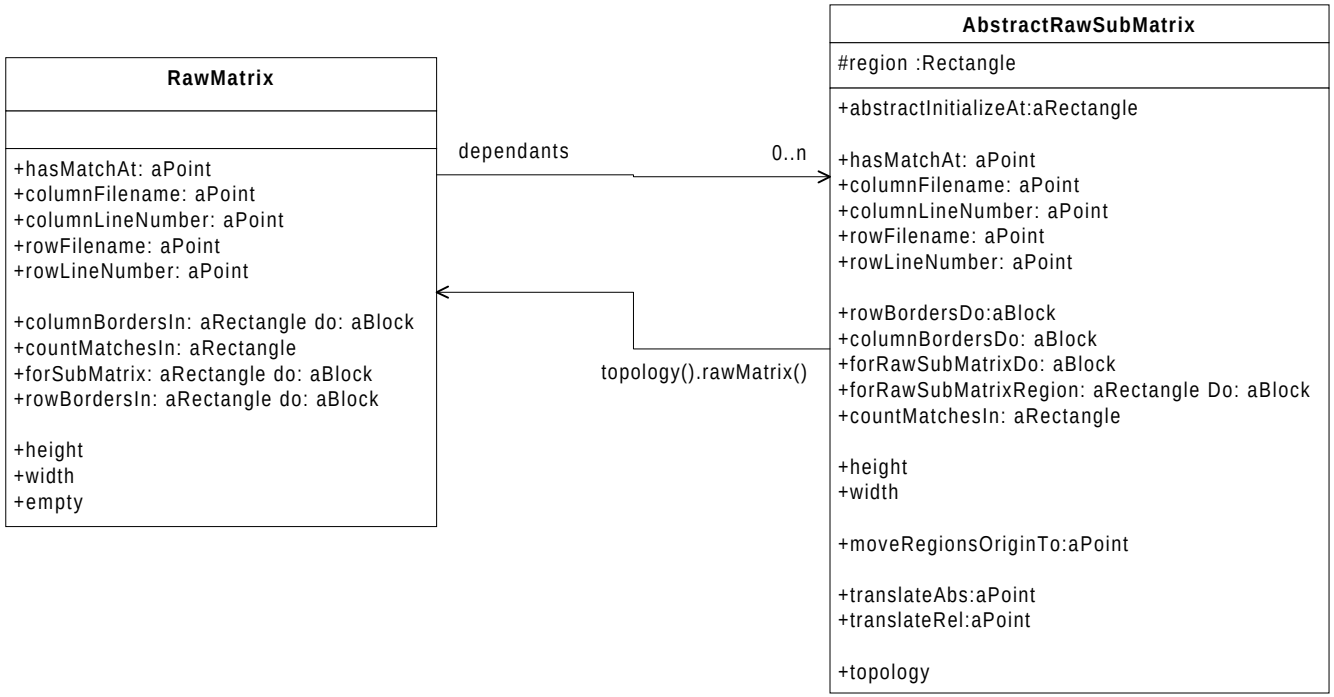


Diagram 2b. *Subclasses of the AbstractRawSubMatrix* class

The *AbstractUserSelection* class is a subclass of the *AbstractRawSubMatrix* class. It represents the *user selection region*. The *AbstractOverView* class and *AbstractSuperOverView* class represent the *overview* respectively *super overview region*. They are subclasses of the *AbstractInformationMuralMatrix* class, which is a subclass of the *AbstractRawSubMatrix* class.

- ***Features of the AbstractInformationMuralMatrix class***:
  - *AbstractInformationMuralMatrix* class stores in the attribute *imMatrix* the *Information Mural matrix*, which represents the match densities of the own *raw sub matrix region*.
  - The *Information Mural matrix* has height *n''* and width *m''*.
    - The upper left corner has co-ordinates *(1,1)*
      (This definition results from the usage of the *TwoDList* class)
    - and the bottom right corner has co-ordinates *(n'', m'')*.
  - The size of each *bin region* is stored in the *binLen* attribute.
- ***Methods summary of the AbstractInformationMuralMatrix class***:
  - The method '**buildImMatrixReportingProgressionTo:aValueHolder PercentageStep:aValue**' builds the *Information Mural matrix* for the covered region.
  - The class provides a set of methods for mapping co-ordinates between the *raw sub matrix region* and the *Information Mural matrix*.
  - It also provides an enumerating method: '**forInformationMuralMatrixDo:aBlock**' evaluates for each *bin value* the block **aBlock**.

The current *raw matrix* is presented by the *DuplocPresentationModel* instance with a *two level* or *three level view representation*. The *DuplocPresentationModel* instance holds three instances: The first is a sort of *AbstractUserSelection*, the second is a sort of *AbstractOverView* and the third is a sort of *AbstractSuperOverView* – the details are described in the third part. The section *5.3.1 Representing a raw matrix* explains the formulas, how the parameters about the *binLen* and the region sizes for each three classes are selected. The class methods and class attributes, which are underlined in the Diagram 2b, implement these described formulas.

The *raw matrix* size changes, according to the loading of new files or deleting of present files, and the contents changes, if lines are 'deleted' in the 'Files compared' window – see section *2.3 Using Duploc in the original Interactive Mode*. Therefore an update protocol between the *RawMatrix* instance and its dependants, which are subclasses of the *AbstractRawSubMatrix* class, is used – see section *5.3.2 RawMatrix class update protocol to its dependants*. These dependants have a defined behaviour to changes – see section *5.3.3 AbstractRawSubMatrix class behaviour to RawMatrix class changes*. If a dependant is a subclass of the *AbstractInformationMuralMatrix* class, then the behaviour must also be extended – see section *5.3.4 The AbstractInformationMuralMatrix class extends the AbstractRawSubMatrix class behaviour*.

Diagram 2b. Subclasses of the *AbstractRawSubMatrix* class

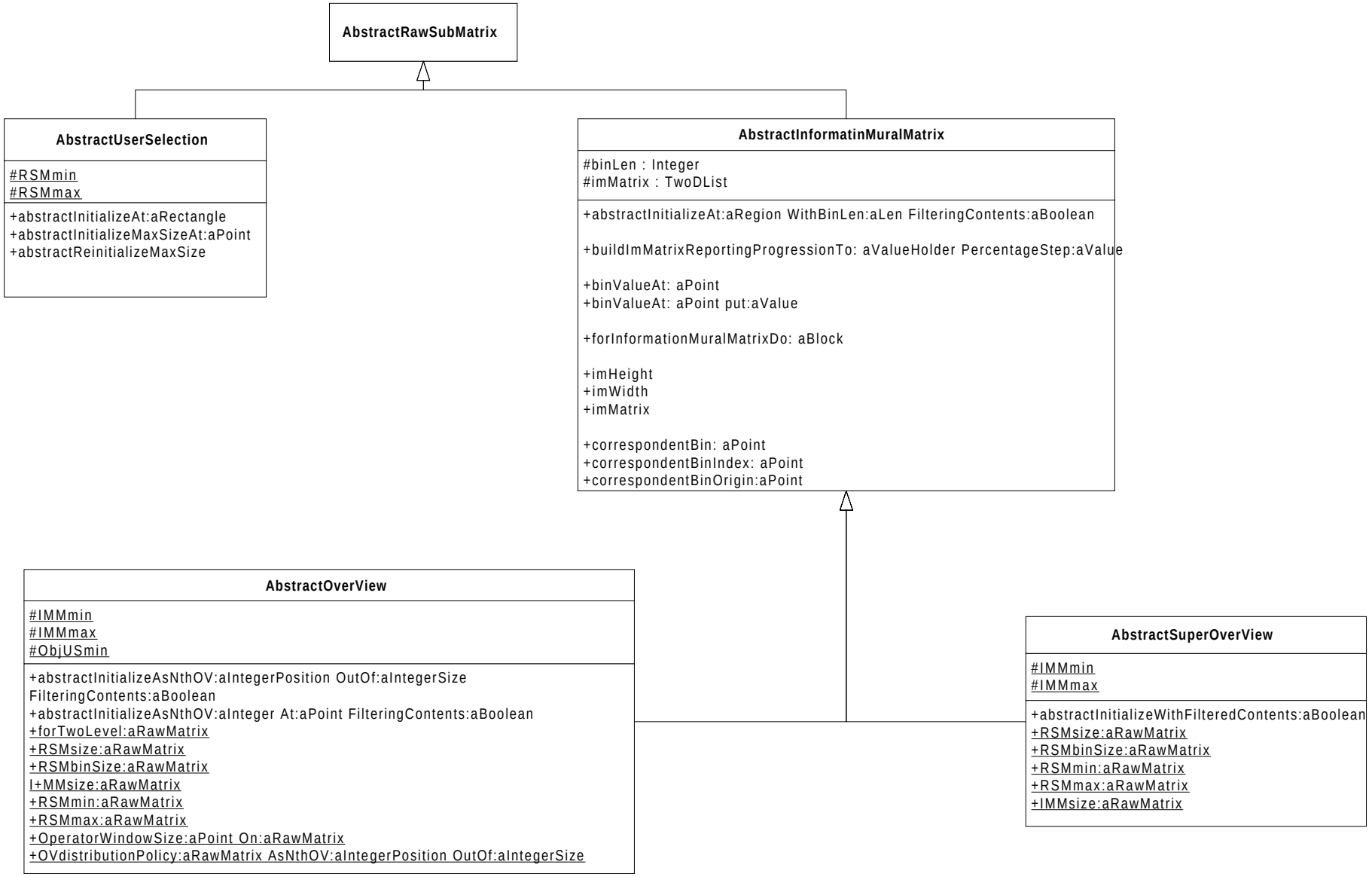


Diagram 3a. Classes, which model the view state, the controller state and the cursor information. They interact with classes, which implement the *RawSubMatrix* class type, and with classes, which store cached data.

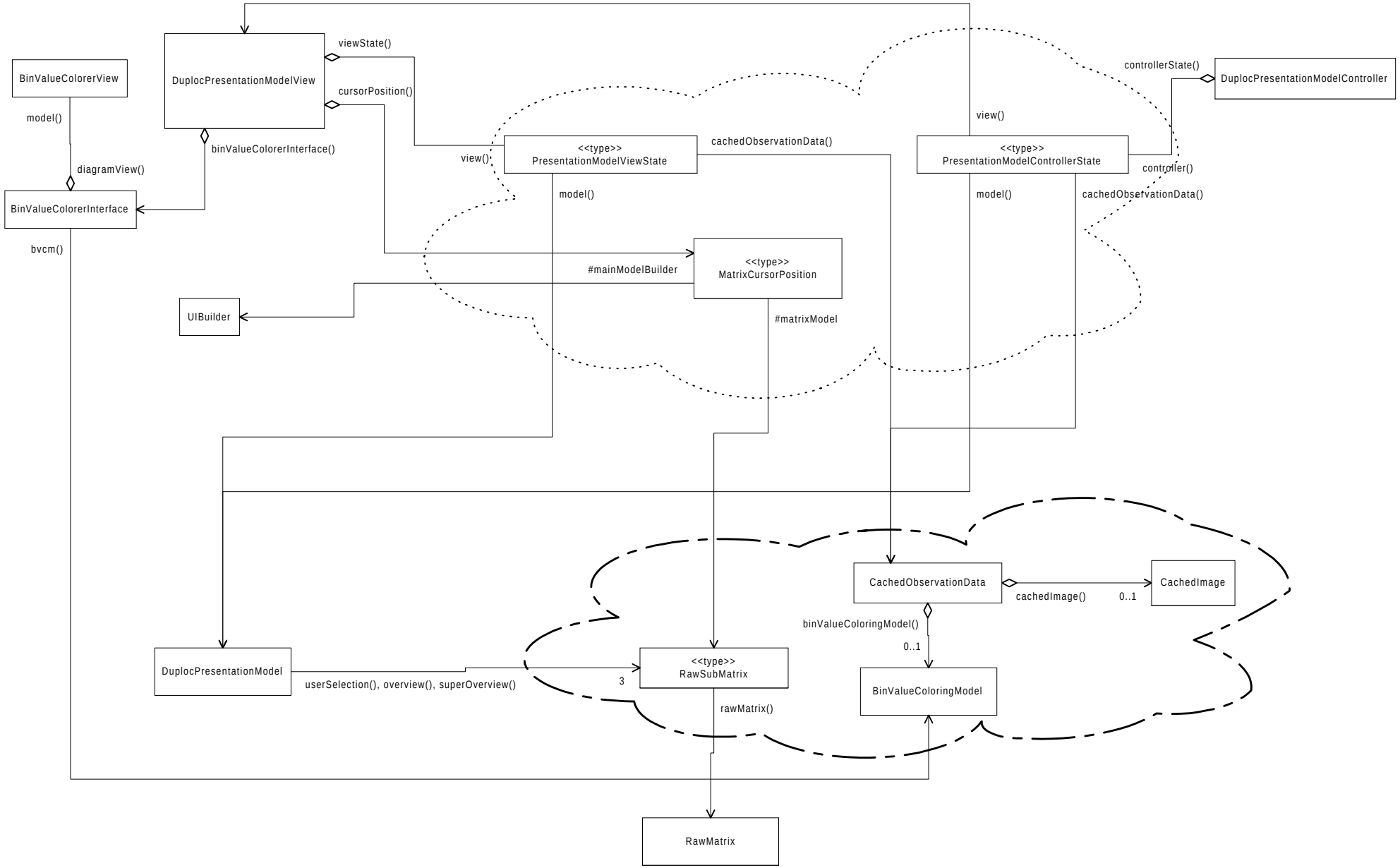


Diagram 3a. ***Classes, which model the view state, the controller state and the cursor information. They interact with classes, which implement the RawSubMatrix class type, and with classes, which store cached data.***

This third part describes the role of the classes inside the *graphical* and *model cloud* introduced in Diagram 1. This Diagram 3a describes the topology of the *graphical cloud*, meaning all the classes and relationships forming the *graphical cloud*, and the concept of the *model cloud*.
As mentioned in the first part, the *DuplocPresentationModelView* class and *DuplocPresentationModelController* class have several states: These states are implemented according to the 'state' design pattern. The role of the classes inside the *graphical cloud* is to represent these different states. The *PresentationModelViewState* type class stands for the set of all classes, inside which each class implements a *DuplocPresentationModelView* state. The *PresentationModelControllerState* type class stands for the set of all classes, inside which each class implements a *DuplocPresentationModelController* state. The *MatrixCursorPosition* type class represents the different implementation classes used to display via the *UIBuilder* class the *raw matrix* information below the current mouse cursor position.
In each application state, the three instances of the implementation classes inside the *graphical cloud* interact with the instances of the classes drawn in the *model cloud*. The *model cloud* consist of a more complex topology than the one sketched in Diagram 3a. The complete topology of the *model cloud* is described in the fourth part.

The role of the classes inside the *model cloud* is to guarantee the different display modes of the application. The application has up to three different display modes:
- *user selection display mode*
- *overview display mode*
- *super overview display mode*

The two display modes *overview* and *super overview* need:
- to store their displayed *binplot* as a cached image, because the creation time of a *binplot* is quite time consuming: At (almost) every position in the *binplot* a grey shaded dot is drawn. Each used grey value is returned by the associated 'bin value colouring function'. Therefore it needs …
- to store the associated 'bin value colouring function' used for creating the latest cached image.

A 'bin value colouring function' is realised with the *BinValueColoringModel* class – see section *5.3.6 The 'bin value colouring model'*.

All three display modes need also to store:
- the current dot size
- the current scrollbar positions
- the previous dot size
- the previous scrollbar positions

This allows the zoom back functionality in each display mode.

Therefore the *model cloud* contains for each display mode:
- An instance of the *CachedObservationData* class storing all data described above.
- An instance of the implementation class for the *RawSubMatrix* class type. This implementation class is a subclass of *AbstractUserSelection*, *AbstractOverView* or *AbstractSuperOverView* class. It is specific to this application topology – see further down explanations about Diagram 3b.

The instances in the upper cloud interact with the instances in the *model cloud*: In order to understand the type of interactions, which occur, the following update scenario in the *overview display mode* is presented:

Scenario: view area must be redrawn
The window system notifies to the *DuplocPresentationModelView* instance to redraw the window view area.
This request is forwarded by the *DuplocPresentationModelView* instance to the current view state instance. Each view state class (see Diagram 3f below) has the role to display inside the window view area the latest image of the *overview binplot* and the orange rectangle representing the *user selection region*.
First the view state instance verifies, if the *CachedObservationData* instance, returned by the referencing method '**cachedObservationData**', has still a cached image. If 'nil' is returned the image was not created yet (lazy instantiation) or it was invalidated – see part four. If the image is not present the view state instance accesses by computing 'self model overview' the instance, which is a subclass of the *AbstractOverView* class, in order to create the *overview binplot* as a new instance of the *CachedImage* class. This image is created with the cached *BinValueColoringModel* instance stored in the *CachedObservationData* instance. The newly created image is stored in the same *CachedObservationData* instance.

Once a valid cached image is available, it is displayed on the window view area.
Finally, in order to display the current position of the orange rectangle the view state instance must access by sending the message 'self model userSelection' the instance, which is a subclass of *AbstractUserSelection*, in order to request the current *user selection region* co-ordinates.

In the *super overview display mode* the view state instance also caches its *super overview binplot* in its correspondent *CachedObservationData* instance. But, in the *user selection display mode* no cached image is currently created; the drawing occurs directly to the screen.

The role of each referencing method in the *graphical cloud* (e.g. cachedObservationData() on Diagram 3a) is to return a reference of the appropriate instance in the *model cloud*. It is obvious, that these referencing methods must know the exact application topology.

The *BinValueColorerInterface* class is the application model of the 'bin colouring tool' window. It always shows the *BinValueColoringModel* instance of the current display mode. After each display mode change the *BinValueColorerInterface* instance is linked with the cached *BinValueColoringModel* instance: It sets the labels and controls the sliders in the window according to the settings in the current referenced *BinValueColoringModel* instance. The *BinValueColorerView* instance draws the resulting mapping function - see section *4.2.5 The bin value colouring function*.

Diagram 3b. ***Classes implementing the RawSubMatrix type class – they are topology dependant subclasses of the AbstractRawSubMatrix*** **class**

The classes implementing the *RawSubMatrix* type class are:
- *UserSelection*
- *Overview*
- *SuperOverview*

All three classes are subclasses of the corresponding abstract classes. They bear the knowledge about the application topology by implementing the method '**topology**', which returns 'self' and the method '**rawMatrix**', which returns the instance variable on the *RawMatrix* instance. As explained in part four, these classes are not directly dependant of the *RawMatrix* instance. (This is the application topology aspect separated from the corresponding abstract classes.)

Diagram 3c. ***Classes implementing the MatrixCursorPosition type class***

The classes implementing the *MatrixCursorPosition* type class are:
- *InformationMuralMatrixCursorPosition* – it is used in the *overview* and *super overview display mode*
- *RawSubMatrixCursorPosition* – it is used in the *user selection display mode*

As shown on Diagram 3a, these classes accesses the *UIBuilder* class of the application model (see also Diagram 1) for displaying on the GUI labels the information about the current cursor position

**Diagram 3b. Classes implementing the *RawSubMatrix* type class -
they are topology dependant subclasses of the *AbstractRawSubMatrix* class**

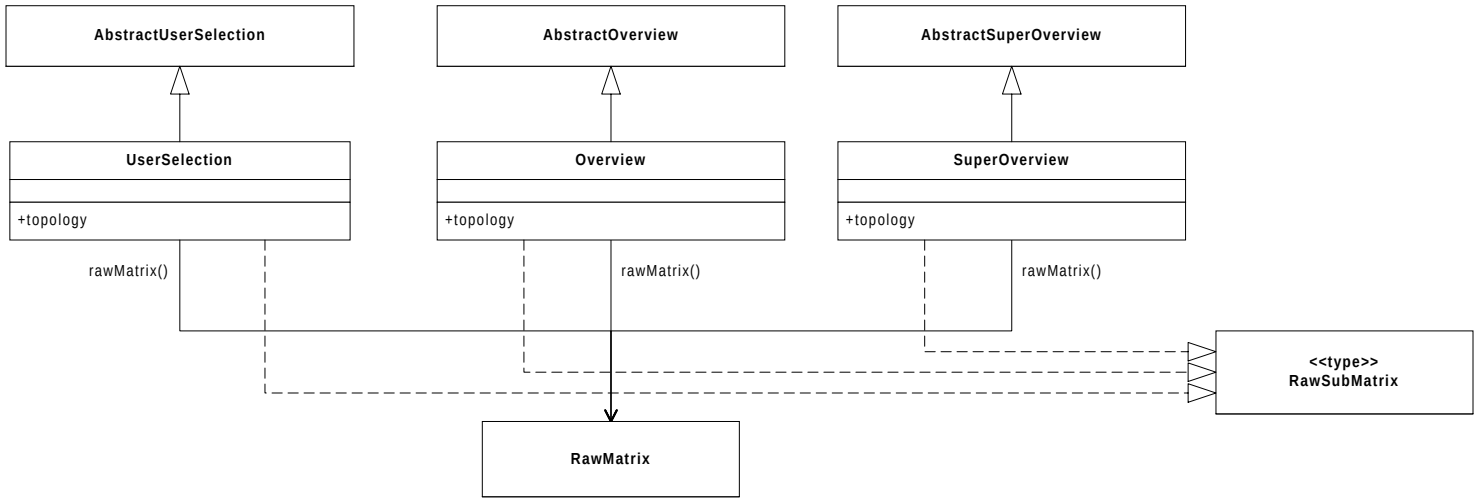


Diagram 3c. Classes implementing the *MatrixCursorPosition* type class

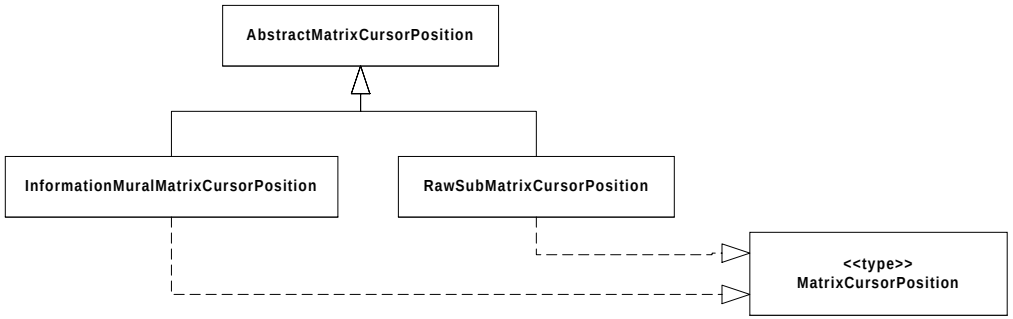


Diagram 3d. ***Classes implementing the PresentationModelViewState type class***

Diagram 3e. ***Classes implementing the PresentationModelControllerState type class***

These two class diagrams show the view state classes implementing the *PresentationModelViewState* type class respectively the controller state classes implementing the *PresentationModelControllerState* type class. (N.B. in Diagram 3e the classes with the round corners represent the implementation classes).

Diagram 3f. ***State diagram of the DuplocPresentationModelView class***

This diagram shows the state diagram of the *DuplocPresentationModelView* class. Each state corresponds with an implementing class in Diagram 3d; only the leading 'PMVS' is omitted from the classname.
(N.B. The events written in normal font are from the mouse menu selection. The events written in italic font are of another nature – the appropriate explanations appear on the drawing.)

Diagram 3g. ***State diagram of the DuplocPresentationModelController class***

This diagram shows the state diagram of the *DuplocPresentationModelController* class. Each state corresponds with an implementing class in Diagram 3e; only the leading 'PMCS' is omitted.
The events are written with the same notation as in Diagram 3f.

Diagram 3h. ***Object instance diagram of the class diagram presented in Diagram 3a showing following situation: The user selected the overview display mode in a three level view raw matrix representation and he wants to reposition the overview region.***

This instance diagram is an example of the class diagram shown in the Diagram 3a. It shows the following application state: The user selected the *overview display mode* in a *three level view representation*. He wants to reposition the *overview region* on the *raw matrix* area.

Diagram 3d. Classes implementing the *PresentationModelViewState* type class

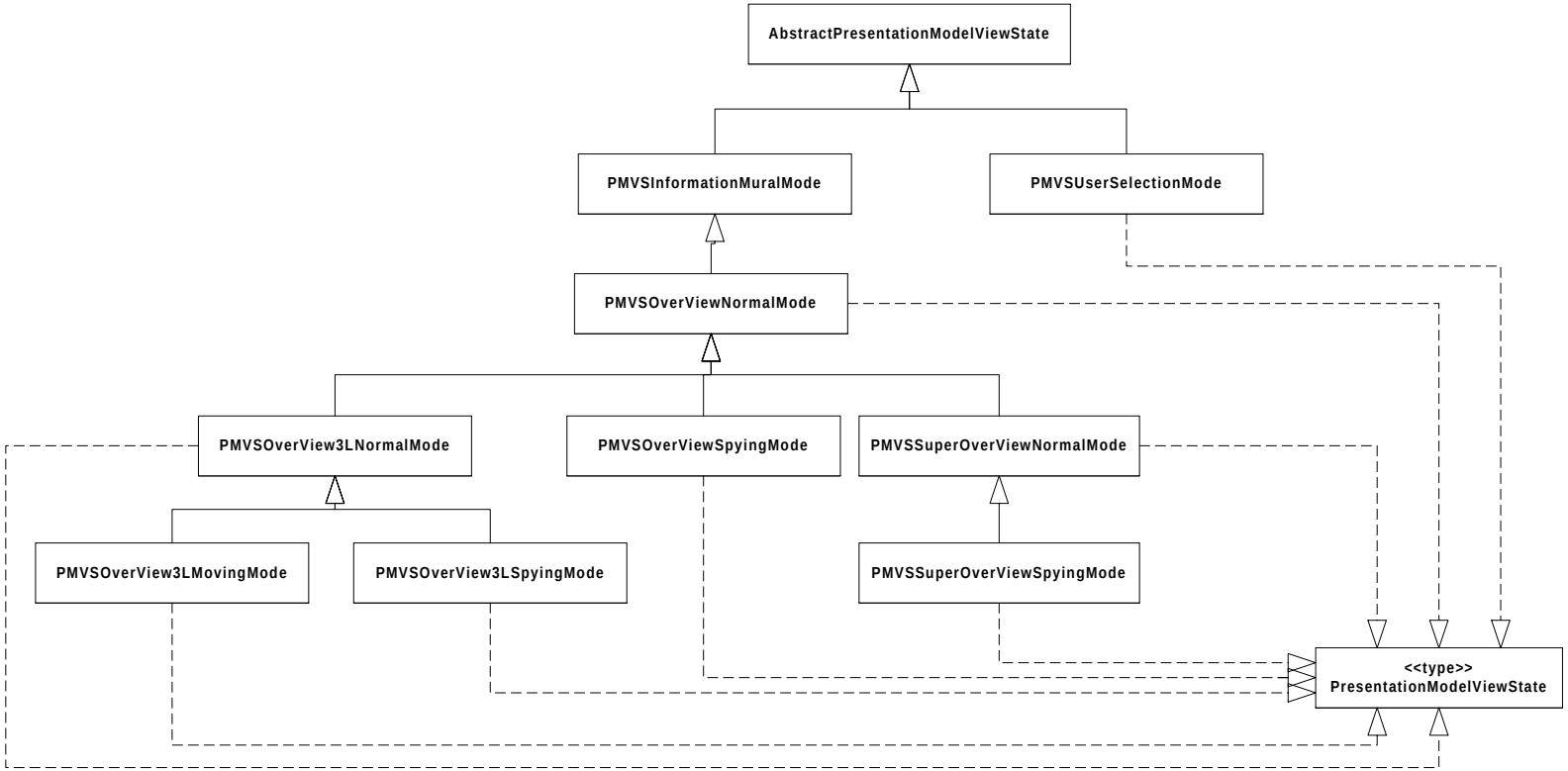
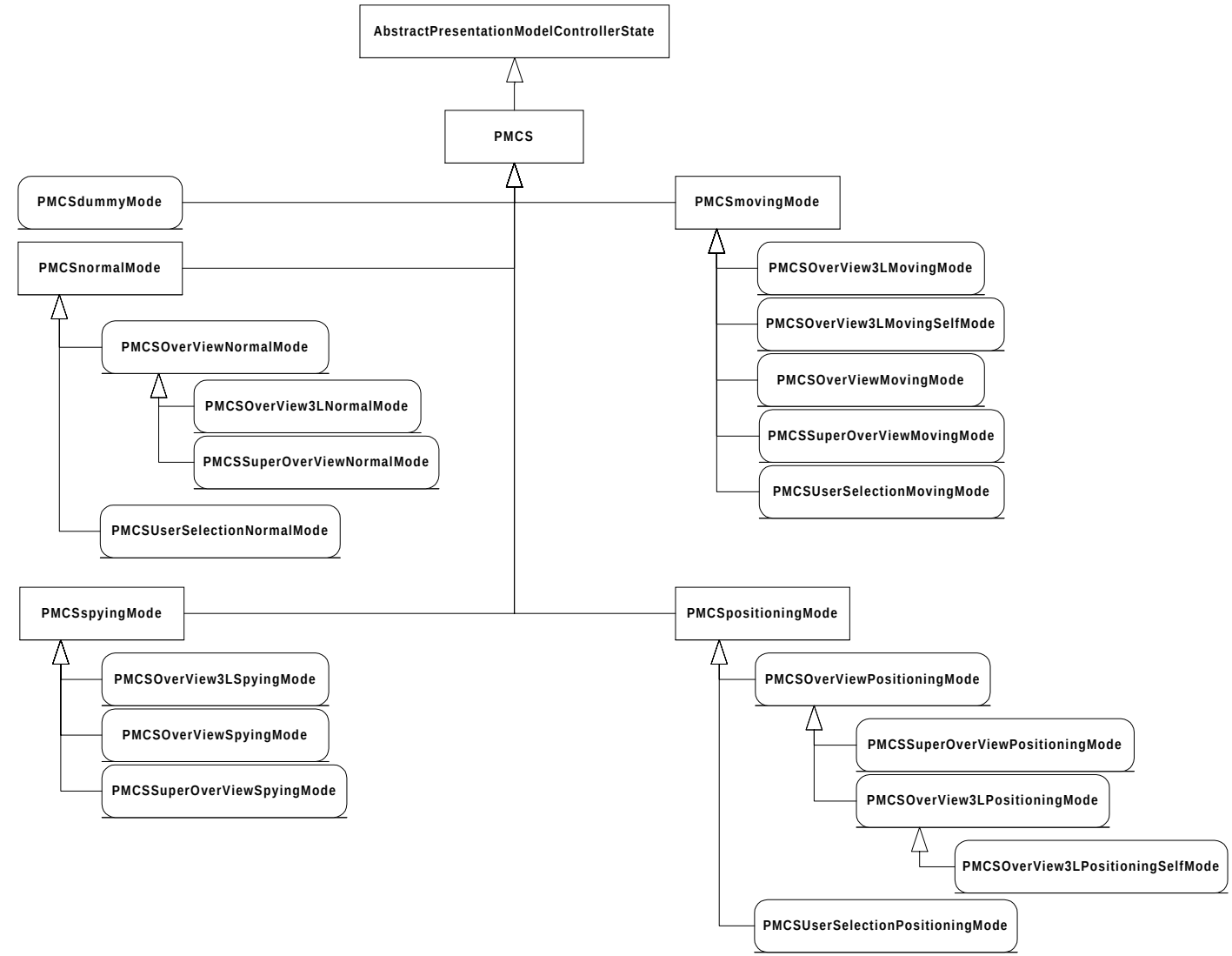


Diagram 3e. Classes implementing the *PresentationModelControllerState* type class



Symbol:

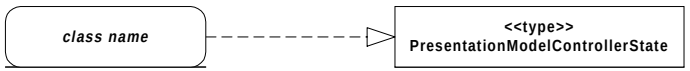
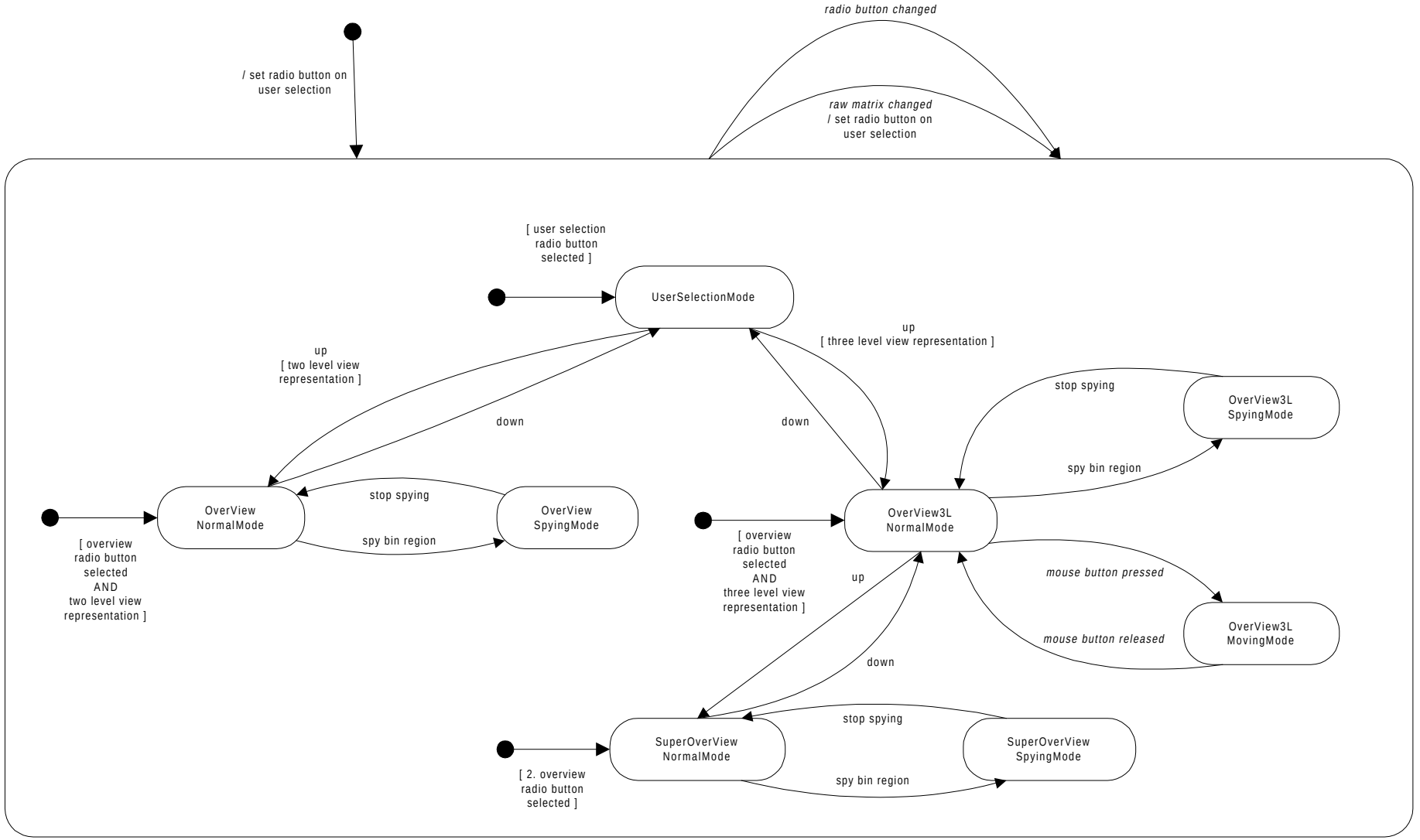


Diagram 3f. State diagram of the *DuplocPresentationModelView* class

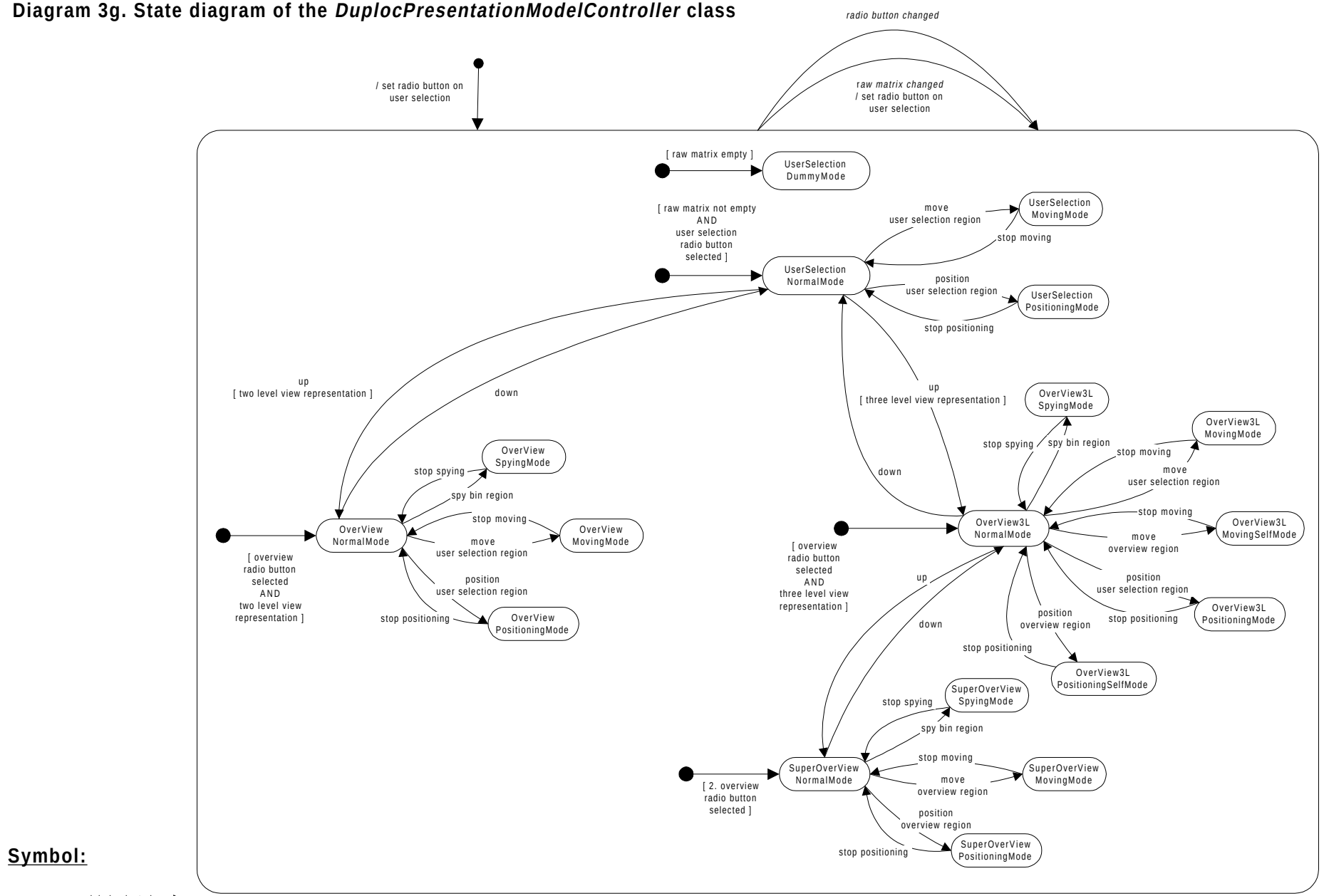


Symbol:

mouse menu label selected →

another type of event →

Diagram 3g. State diagram of the *DuplocPresentationModelController* class



Symbol:

mouse menu label selected

another type of event

Diagram 3h. Object instance diagram of the class diagram presented in Diagram 3a showing following situation:
The user selected the *overview display mode* in a *three level view raw matrix representation* and he wants to reposition the *overview region*.

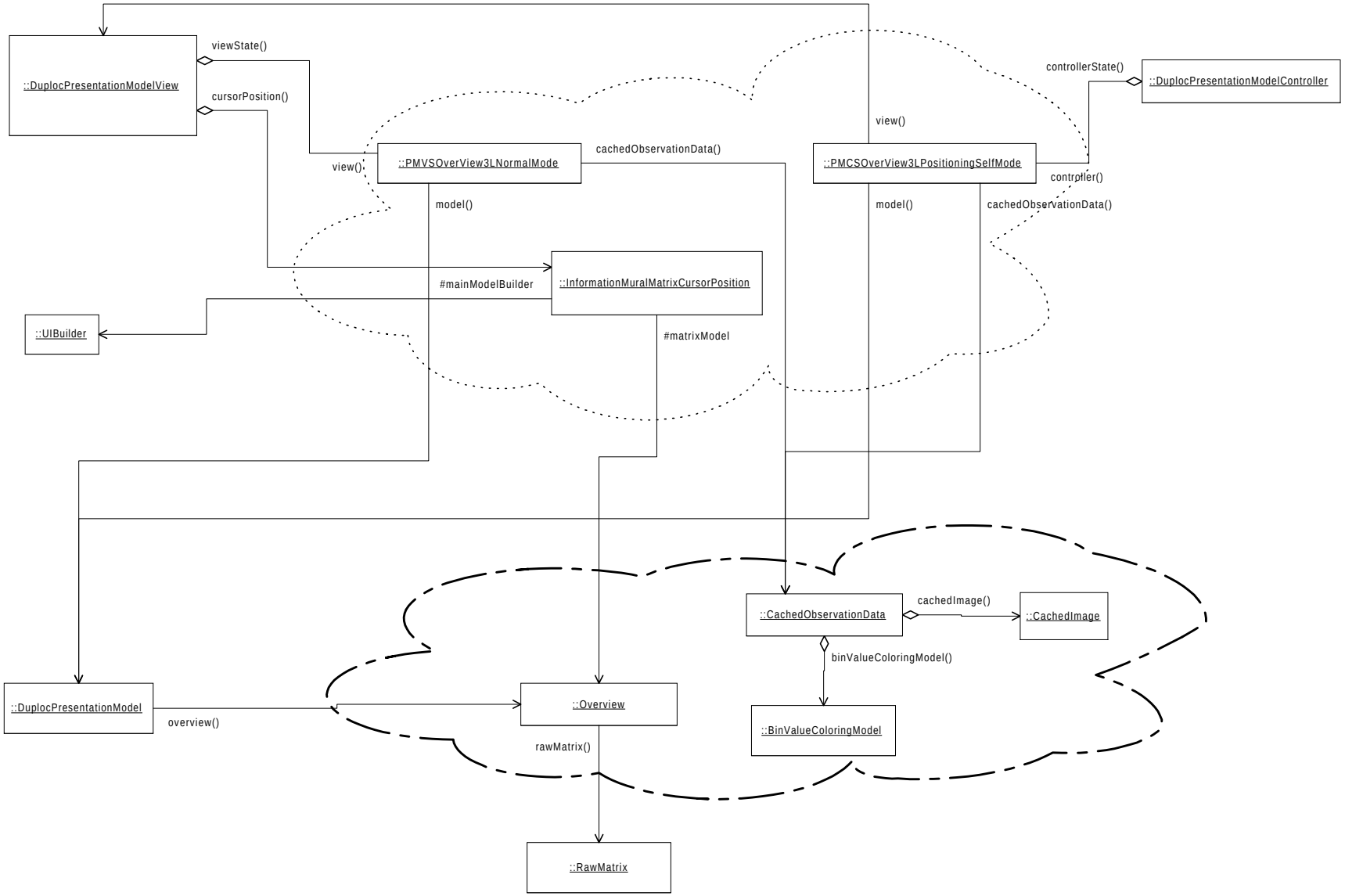


Diagram 4a. ***Classes used by the DuplocPresentationModel class for presenting the current raw matrix***

This diagram describes the topology of the *model cloud*:

- The *DuplocPresentationModel* instance references up to 3 *ObservationOnRawSubMatrix* instances. The *UserSelection*, *Overview* and *SuperOverview* instances are each attached to an *ObservationOnRawSubMatrix* instance.
- The *ObservationOnRawSubMatrix* instance is capable to reference a set of *CachedObservationData* instances, but currently only up to one is referenced: The number of *ObservationOnRawSubMatrix* instances corresponds with the number of possible display modes (representation levels). If the application supports several view areas or windows simultaneously, the case would occur, where two view areas show simultaneously the same display mode, but with a different dot size. Therefore the number of *CachedObservationData* instances corresponds with the number of GUI view areas managed by the application – the current *Duploc* application manages one such view area. Therefore each *ObservationOnRawSubMatrix* instance references maximal one *CachedObservationData* instance. The *ObservationOnRawSubMatrix* instance controls the validity of the stored *CachedObservationData* instance(s).
- The *DuplocPresentationModel* instance is dependant of each three *ObservationOnRawSubMatrix* instances.

The topology is explained with the update propagation in two different scenarios:

Scenario 1: The user selection region is moved or (re)positioned
If the *UserSelection* instance receives from the controller state instance the message to move or (re)position the *user selection region* then the instance informs its dependant – the *ObservationOnRawSubMatrix* instance.
The *ObservationOnRawSubMatrix* instance informs each *CachedObservationData* instance to invalidate the cached image, and updates its dependant – the *DuplocPresentationModel* instance.
The *DuplocPresentationModel* instance informs its dependant – the *DuplocPresentationModelView* instance.
The *DuplocPresentationModelView* instance invalidated the GUI view area, which causes the window system to notify to the same *DuplocPresentationModelView* instance to redraw the view area – see the described scenario in the third part.

Scenario 2: Changes to the raw matrix
If the *RawMatrix* instance broadcasts the update protocol (discussed in the section *5.3.2 RawMatrix class update protocol to its dependants*) it is important, that the following sequence is guaranteed:

1. all instances implementing the *RawSubMatrix* type class are updated
2. the *DuplocPresentationModel* instance is updated.

This is the reason, why the *UserSelection*, *Overview* and *SuperOverview* instances are not directly dependant of the *RawMatrix* instance. The *DuplocPresentationModelProtocolTransformer* instance is dependant of the *RawMatrix* instance: It is responsible to forward each update message from the *RawMatrix* instance to its dependants, the instances implementing the *RawSubMatrix* type class. Once each dependant is updated, it sends to the 'transformedProtocolReceiver', the *DuplocPresentationModel* instance, the update messages to adapt the representation model (*two* or *three level view representation*) and to update the view area.
Therefore in this update sequence scenario each *ObservationOnRawSubMatrix* instance will not update its dependant, the *DuplocPresentationModel* instance, after invalidating the referenced *CachedObservationData* instance. If it would update the *DuplocPresentationModel* instance the GUI would be unnecessarily updated several times.

Diagram 4a. Classes used by the *DuplocPresentationModel* class for presenting the current *raw matrix*

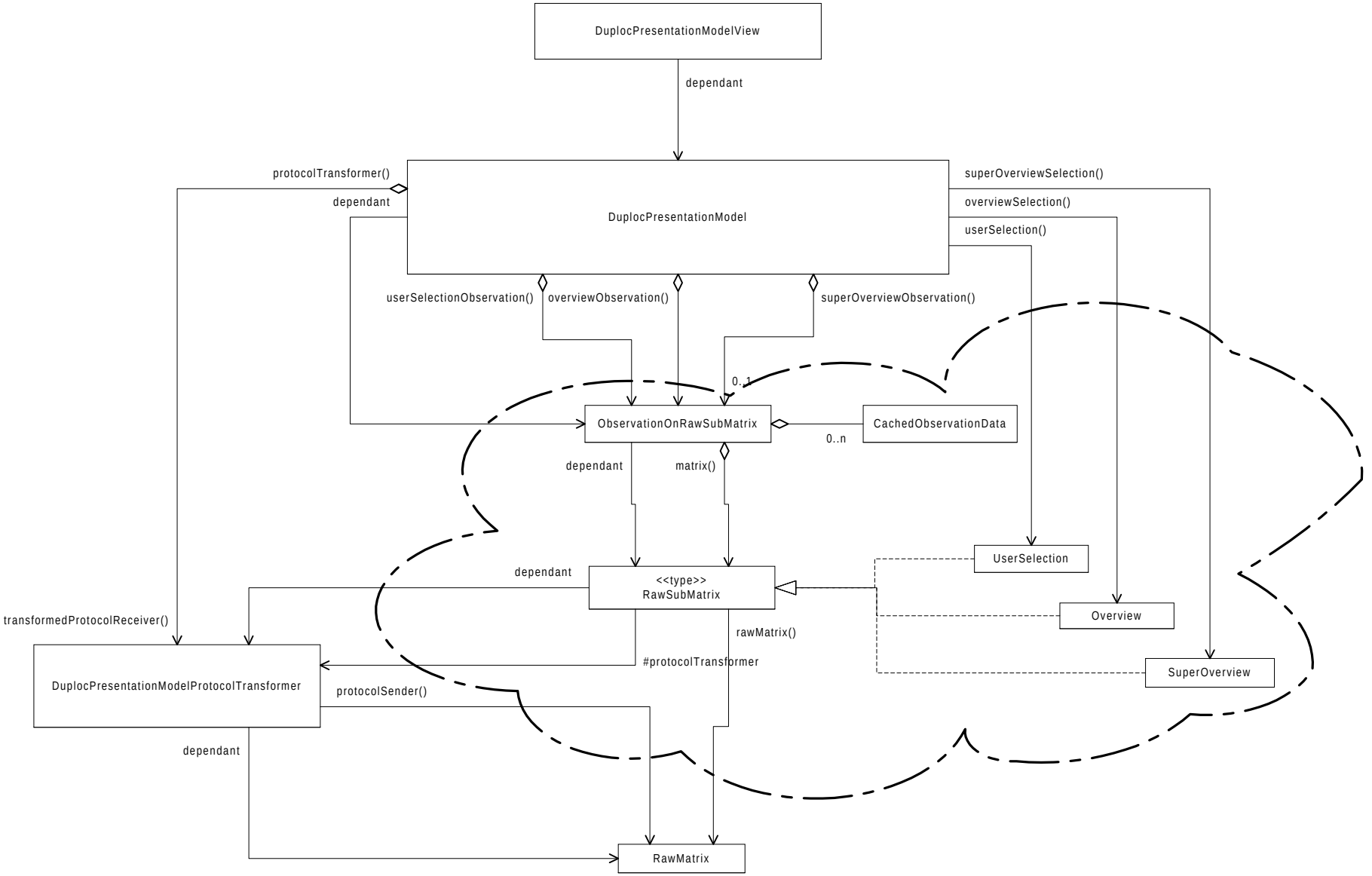


Diagram 4b. ***Object instance diagram of the class diagram presented in Diagram 4a showing the objects presenting the current RawMatrix instance in a three level view representation.***

This instance diagram is an example of the class diagram shown in Diagram 4a. It shows a *three level view representation* of the current *RawMatrix* instance.

Diagram 4c. ***A collection of cached objects for each 'view' on the same RawSubMatrix type class***

This class diagram shows some details of the *ObservationOnRawSubMatrix* and *CachedObservationData* classes. Each *CachedObservationData* instance has an assigned id – see the attribute *id*. With the method '**cachedObjectWithId:aNumber**' the *ObservationOnRawSubMatrix* instance returns a reference on the *CachedObservationData* instance with the id 'aNumber'.

Diagram 4d. ***ForwardingObject, ProtocolTransformer & DuplocPresentationModelProtocolTransformer classes***

This class diagram shows the super class of the *DuplocPresentationModelProtocolTransformer* class: The *ForwardingObject* class forwards each received update message to its dependants. The *ProtocolTransformer* class extends this behaviour by giving the possibility to its subclasses to intervene on an update protocol: The update protocol sent by the *protocol sender* is forwarded, unless the corresponding '**update:anAspectSymbol …**' method is overwritten. Like this, the *transformed protocol receiver* instance can be notified. The *ProtocolTransformer* class holds these two references. The *DuplocPresentationModelProtocolTransformer* class implements the role discussed above by using these two references – see section about Diagram 4a.

Diagram 4b. Object instance diagram of the class diagram presented in Diagram 4a showing the objects presenting the current *RawMatrix* instance in a *three level view representation*.

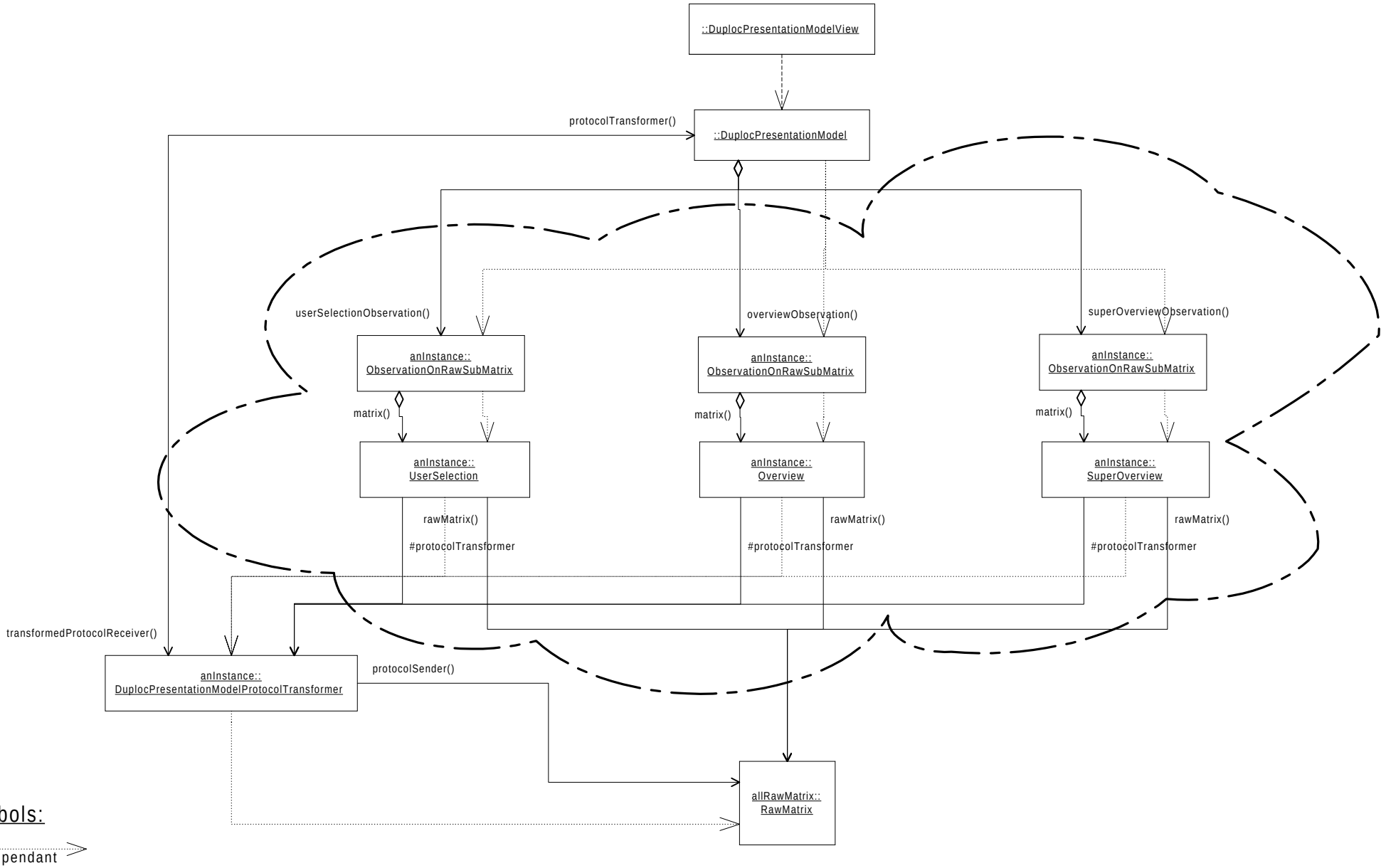


Diagram 4c. A collection of cached objects for each 'view' on the same *RawSubMatrix* type class

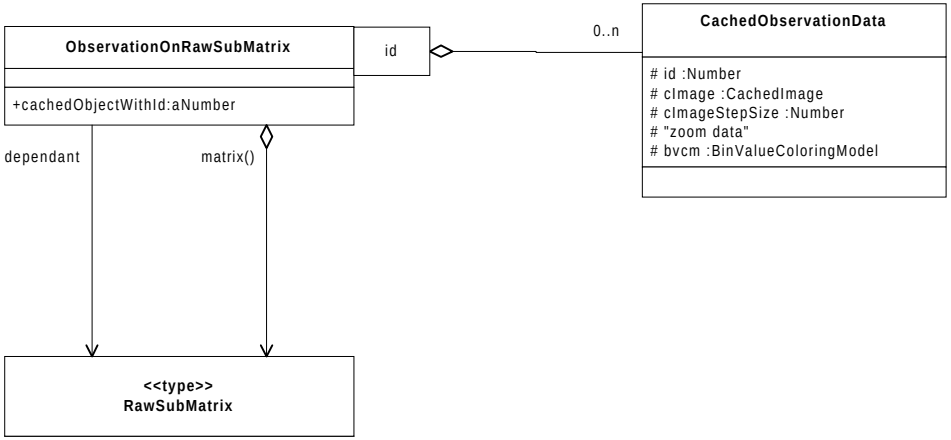
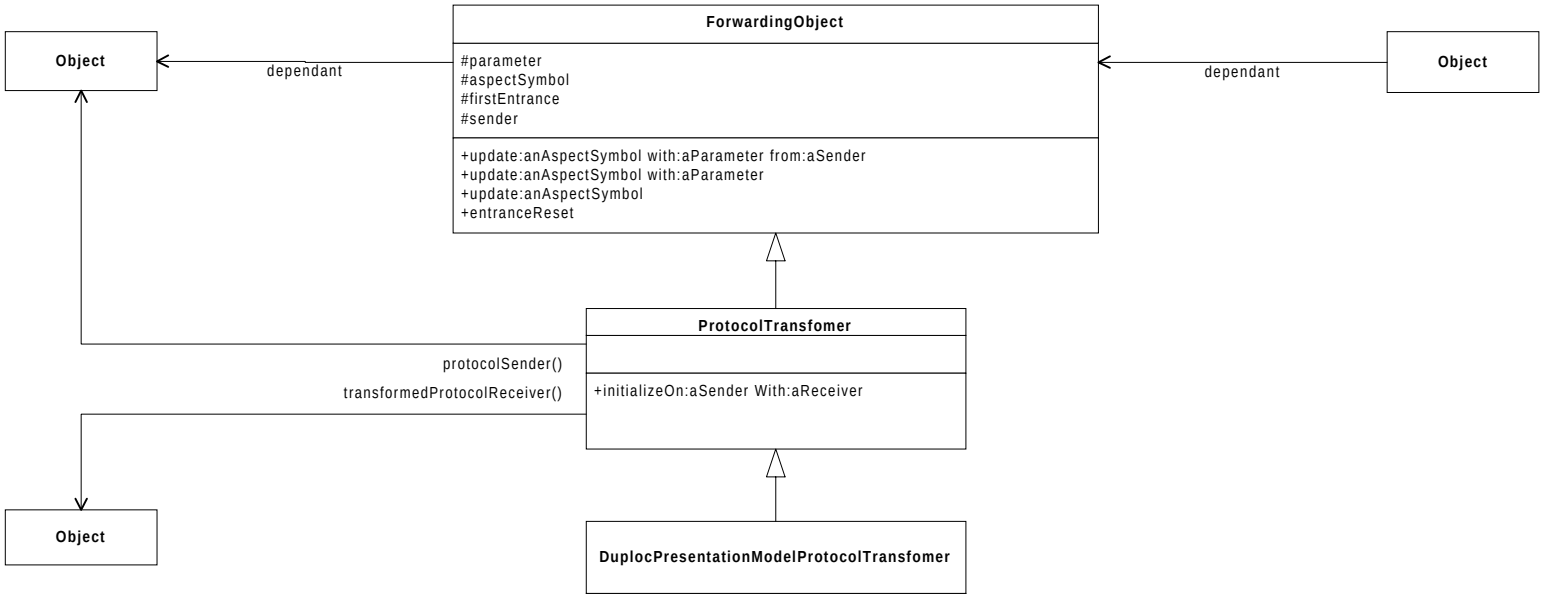


Diagram 4d. *ForwardingObject*, *ProtocolTransformer* & *DuplocPresentationModelProtocolTransformer* classes



5.3 System details

5.3.1 Representing a raw matrix

5.3.1.1 Introduction

This section describes, how the *user selection*, *overview* and *super overview region* sizes are related, in order to achieve the best possible *raw matrix* representation – see section *4.1.2 Representing a large raw matrix.*

5.3.1.2 Defined abbreviations

The important data-structures are referenced with the following abbreviations:

- **rm** – *raw matrix*
- **us** – *user selection*
- **ov** – *overview*
- **sov** – *super overview*

The us, ov and sov are considered a sort of:

- **rsm** – *raw sub matrix*

An *Information Mural Matrix* has this abbreviation:

- **imm** – *information mural matrix*

The *raw matrix* has the following attributes

Abbreviations	***Description***	***Const/Variable***
rm.rsm.size	size of the *raw matrix*	variable

The us, ov and sov have the following attributes:

us:

Abbreviations	***Description***	***Const/Variable***
us.rsm.min	minimal size of the *raw sub matrix* representing the *user selection region*	const
us.rsm.default	default size of the *raw sub matrix* representing the *user selection region*	const
us.rsm.max	maximal size of the *raw sub matrix* representing the *user selection region*	const
us.rsm.size	size of the *raw sub matrix* representing the *user selection region*	variable

ov:

Abbreviations	***Description***	***Const/Variable***
ov.rsm.min	minimal size of the *raw sub matrix* representing the *overview region*	variable
ov.rsm.max	maximal size of the *raw sub matrix* representing the *overview region*	variable
ov.rsm.size	size of the *raw sub matrix* representing a section or the complete *overview region*	variable
ov.rsm.bin.size	size of each *bin region* – all *bin regions* partition a section or the complete *overview region*	variable
ov.imm.min	minimal size of the *information mural matrix* representing the match densities in the *overview region*	const
ov.imm.max	maximal size of the *information mural matrix* representing the match densities in the *overview region*	const
ov.imm.size	size of the *information mural matrix* representing the match densities in the *overview region*	variable
ov.obj-us.size	size of the orange rectangle in dots in the *overview binplot* representing the *user selection region*	variable
ov.obj-us.min	minimal size in dots of the orange rectangle in the *overview binplot* representing the *user selection region*, which is allowed in a *two level view representation*	const

sov:

Abbreviations	***Description***	***Const/Variable***
sov.rsm.min	minimal size of the *raw sub matrix* representing the *super overview region*	variable
sov.rsm.max	maximal size of the *raw sub matrix* representing the *super overview region*	variable
sov.rsm.size	size of the *raw sub matrix* representing the *super overview region*	variable
sov.rsm.bin.size	size of each *bin region* – all *bin regions* partition the *super overview region*	variable
sov.imm.min	minimal size of the *information mural matrix* representing the match	const

	densities in the *super overview region*	
sov.imm.max	maximal size of the *information mural matrix* representing the match densities in the *super overview region*	const
sov.imm.size	size of the *information mural matrix* representing the match densities in the *super overview region*	variable
sov.obj-us.size	size of the orange rectangle in dots in the *super overview binplot* representing the *user selection region*	variable
sov.obj-ov.size	size of the blue rectangle in dots in the *super overview binplot* representing the *overview region*	variable

5.3.1.3 Two level / three level view representation selection criteria

As discussed above, once the *raw matrix* grows too big, the orange rectangle representing the *user selection region* in the *overview binplot* would have to be drawn around a fraction of a single *bin* dot. This was the reason for introducing the *three level view representation.*

The attribute **ov.obj-us.min** defines the minimal number of dots, above which the size of the orange rectangle in the *overview binplot* in a *two level view representation* is acceptable – below this threshold a *three level view representation* must be used:

(ov.obj-us.size < ov.obj-us.min)

True: use *three level view representation*

False: use *two level view representation*

5.3.1.4 Attribute value definitions

This section describes, how the us, ov and sov are related in their attribute values to achieve the best possible rm representation. The next section describes the used concept for defining the size of the ov in a *three level view representation* :

- ***remark:*** The two used italic expressions '*two levels*' and 'three *levels*' refer to the *two level* and *three level view representation*.

- ***Auxillary functions:***
  - $\lceil x \rceil := \text{trunc}(x + 1)$
  - $\max_{(x,y)}((x,y)) := \max(x,y)$ of point (x,y).

- ***us:***
  - us.rsm.min := 200x200 , unless rm is smaller.
  - us.rsm.default := 400x400
  - us.rsm.max := 800x800
  - us.rsm.size := selectable by the user inside the allowed range.

- ***ov:***
  - ov.imm.min := 5x5
  - ov.rsm.min := ov.imm.min · ov.rsm.bin.size
  - ov.imm.max := 100x100
  - ov.rsm.max := ov.imm.max · ov.rsm.bin.size
  - ov.imm.size :=
    two levels: $\lceil$ov.rsm.size / ov.rsm.bin.size$\rceil$
    three levels: ov.imm.max
  - ov.rsm.size :=
    two levels: rm.rsm.size
    three levels: ov.imm.size · ov.rsm.bin.size
  - ov.rsm.bin.size :=
    two levels: $\max(2, \max_{(x,y)}(\lceil$ov.rsm.size / ov.imm.max$\rceil))$
    three levels: $\lceil$ sqrt((us.rsm.min / ov.imm.size) · sov.rsm.bin.size) $\rceil$
  - ov.obj-us.size := $\lceil$ us.rsm.size / ov.rsm.bin.size $\rceil$
  - ov.obj-us.min := 10x10

- ***sov:***
  - sov.imm.min := 5x5
  - sov.rsm.min := sov.imm.min · sov.rsm.bin.size

- sov.imm.max := 100x100
- sov.rsm.max := sov.imm.max · sov.rsm.bin.size
- sov.imm.size := ⌈sov.rsm.size / sov.rsm.bin.size⌉
- sov.rsm.size := rm.rsm.size
- sov.rsm.bin.size := max(2, $\max_{(x,y)}$(⌈sov.rsm.size / sov.imm.max⌉))
- sov.obj-us.size := ⌈ us.rsm.size / sov.rsm.bin.size ⌉
- sov.obj-ov.size := ⌈ ov.rsm.size / sov.rsm.bin.size⌉

5.3.1.5 Three level view representation concept

In a three level view representation the question to solve is, how must be the *overview region* selected? The chosen concept is to select the largest possible *information mural matrix* size (ov.imm.size := ov.imm.max) and choose an appropriate *bin* size. What is an appropriate ov.rsm.bin.size (*overview bin* size) ? If the ov.rsm.bin.size is too big, then the sov.obj-ov.size (the size of the 'blue rectangle') appears bigger, but the ov.obj-us.size (the size of the 'orange rectangle') would appear too small and vice versa.
The chosen solution is to have a blue rectangle on the *super overview binplot*, which has an equivalent size to the smallest allowed red rectangle on the *overview binplot* :

(1) **r1 = r2** **,** with
r1 := sov.obj-ov.$size_{min}$:= (ov.imm.size · ov.rsm.bin.size) / sov.rsm.bin.size
(n.b. ov.imm.size := ov.imm.max)
r2 := ov.obj-us.$size_{min}$:= us.rsm.min / ov.rsm.bin.size

The equation (1) can be rewritten in (1'), so that the variable x appears on both sides – x stands for the variable ov.rsm.bin.size:

(1') k1 · x = k2 / x , with
k1 := ov.imm.size / sov.rsm.bin.size
k2 := us.rsm.min
x := ov.rsm.bin.size

By solving the equation (1') we obtain:

(1'') **ov.rsm.bin.size :=** x = sqrt(k2/k1) = **sqrt((us.rsm.min/ov.imm.size) · sov.rsm.bin.size)**

5.3.2 RawMatrix class update protocol to its dependants

The size of the current *raw matrix* represented by the correspondent *RawMatrix* instance varies depending on the current compared files. For any dependants to this model this means, that the *raw matrix* size varies by adding or removing 'strips'. Each 'strip' has the height respectively the width of the current *raw matrix*. These changes are sent to the dependants of the corresponding *RawMatrix* instance according to a defined protocol: Following messages are sent:

- 1. **repairStart**
- 2. Sequence of **insertArea** and/or **removeArea**
- 3. **repairNow**

The first and last message (see 1. and 3.) have the purpose of synchronisation. The sequence of messages in-between (see 2.) has the purpose to notify the inserted or removed areas ('strips'). Each message sends therefore a parameter *aRectangle*, which specifies the concerned area.
The contents of the current *raw matrix* can also vary. e.g. The user might 'delete' a line in the '*Files compared*' window – see section *2.3 Using Duploc in the original Interactive Mode*. Following message is sent by the *RawMatrix* instance:

- **filteredContents**

It was also defined as a future extension, that any client of this class must specify, if it wants to access to the contents of the *raw matrix* by taking or not by taking in account the 'deleted lines' – currently this feature is not supported by the *RawMatrix* class.

5.3.3 *AbstractRawSubMatrix* class behaviour to RawMatrix class changes

5.3.3.1 Adaptation behaviour concept

Each *AbstractRawSubMatrix* instance is a dependant of a *RawMatrix* instance. Depending on the changes of the *raw matrix* each *raw sub matrix* must have a defined behaviour.

Defined behaviour concept: A *raw sub matrix* keeps covering the region previous to the *raw matrix* changes, by clipping its region and moving its origin point depending on the *raw matrix* changes. Therefore newly added areas are not covered by the *raw sub matrix*.

Here is the list of cases, for which a *AbstractRawSubMatrix* class must have in implemented reaction, in order to fulfil the defined behaviour concept:

- Some abbreviations:
  V/H - … Vertical/Horizontal strip …
  I - … having an Intersection with current *raw sub matrix* …
  L/R - … to the Left/Right of the current *raw sub matrix*.
  T/B - … to the Top/Bottom of the current *raw sub matrix*.
  S - … which splits current *raw sub matrix*
  A/D - Added/Deleted …
- Vertical strips:
  - Vertical strips, which have no intersection with the *raw sub matrix*:
    - AVL - means **A**dded **V**ertical strip to the **L**eft of the current *raw sub matrix*.
    - DVL - means **D**eleted **V**ertical strip to the **L**eft of the current *raw sub matrix*.
    - AVR
    - DVR
  - Vertical strips, which have an intersection with the *raw sub matrix*:
    - DVIL
    - DVIR
    - AVS
    - DVS
- There are 8 equivalent cases for horizontal strips: AHT, DHT, AHB, DHB, DHIT, DHIB, AHS and DHS.

Each of these 16 cases must have a defined reaction. Here a summary, how the *AbstractRawSubMatrix* class reacts to each case:

- AVR, DVR, AHB, DHB
  This will neither affect the origin nor the size of the *raw sub matrix*.
- AVL, DVL, AHT, AHB
  This will only affect the origin of the *raw sub matrix*. The origin x respectively y position will 'increase', if a strip is added, and 'decrease', if a strip is deleted.
- DVIL, DHIT
  This will affect the origin and size of the *raw sub matrix*. The behaviour can be described in two steps:
  1.) First we create an intermediate *raw sub matrix region*. The original *raw sub matrix region* is clipped with the affected 'strip', so that the non intersecting area of the region is kept. This means, that the origin x respectively y position will 'increase', but the corner point (bottom right point) is not changed. The obtained intermediate region has no intersection with the strip.
  2.) Because the new obtained region has no intersection with the strip, the effective deletion corresponds with the cases DVL and DHT mentioned above applied on the new intermediate region.
- DVIR, DHIB
  This will affect only the size of the *raw sub matrix*. The original *raw sub matrix region* is clipped with the affected 'strip', so that the non intersecting area of the region is kept. This means, that the corner x respectively y position will 'decrease', but the origin point is not changed.
- AVS, DVS, AHS, DHS
  These four cases need an extended definition of behaviour. The simplest definition is, that *raw sub matrix region* is split in two *raw sub matrix regions* and that only the left respectively the top *raw sub matrix region* is kept.

5.3.3.2 Received update and sent changed protocol

- **Implementation remarks about the update/changed methods**[15]:
  - Any change in the *AbstractRawSubMatrix* instance invokes first a method named ***anAspectSymbol***, which itself invokes the method ***changeDanAspectSymbol*** ('changeD' followed by 'anAspectSymbol').
  - A subclass must overwrite the method ***anAspectSymbol*** and if necessary invoke itself the method ***changeDanAspectSymbol*** or invoke the method in the super class.
  - An interested dependant must implement the ***update:*** method, which could react with a correspondent method ***updatEanAspectSymbol***.
- **Behaviour** of the *AbstractRawSubMatrix* instance **to** the following **updateD… messages** from the *RawMatrix* instance:
  - During a *RawMatrix* change[16]:
    - **…repairStart**: The current **region** is stored in the attribute **previousRegion**.
    - **…insertArea**: It applies described behaviour in the section *5.3.3.1 Adaptation behaviour concept*.
    - **…deleteArea**: It applies described behaviour in the section *5.3.3.1 Adaptation behaviour concept.*

[15] see in the source code
[16] see section *5.3.2 RawMatrix class update protocol to its dependants*

- **…repairNow**: Depending on the differences between the current **region** compared to the **previousRegion** following messages are sent
  - **observedContentsHasNewLocation**: The origin was moved.
  - **regionClipped**: The region size changed.
  - **destroyed**: The region is not valid anymore[17] - see further down **destroyed**.
  - **repairFinished**: All the changes were broadcasted.
  - n.b. **observedContentsHasNewLocation** and **regionClipped** can both be sent.
- If the user has modified the contents of the 'deleted lines container':
  - **…filteredContents**: The message is 'forwarded' to the subclass and/or to the dependant classes.

- **Invoked methods**, **which trigger** a correspondent **changeD… message**, if … :
  - if the region has been released then all references to this instance must be released: **destroyed**
  - if the observed region has changed, but the size remains the same: **regionMoved**
  - if a complete new region was specified: **regionRedefined**

5.3.4 The AbstractInformationMuralMatrix class extends the *AbstractRawSubMatrix* class behaviour

5.3.4.1 Received update and sent changed protocol

- **Behaviour** of the *AbstractInformationMuralMatrix* instance **to** the following **updateD… messages** from the *RawMatrix* instance:
  - During a *RawMatrix* instance change[18]:
    - **…repairStart**: nothing to do.
    - **…insertArea**: nothing to do.
    - **…deleteArea**: nothing to do.
    - **…repairNow**: nothing to do.
  - If the user has modified the contents of the 'deleted lines container':
    - **…filteredContents**: If the matrix has interest in a filtered contents, then the instance variable *imMatrix* is destroyed.
- **Behaviour** of the *AbstractInformationMuralMatrix* instance **to** the following **updateD… messages** from the *AbstractRawSubMatrix*:
    - **...observedContentsHasNewLocation**: -.
    - **...regionClipped**: The instance variable *imMatrix* is destroyed.
    - **...destroyed**: The instance variable *imMatrix* is destroyed.
    - **...repairFinished**: nothing to do.
    - **...regionMoved**: The instance variable *imMatrix* is destroyed.
    - **...regionRedefined**: The instance variable *imMatrix* is destroyed.
- **Invoked methods**, **which trigger** a correspondent **changeD… message**, if … :
  - if the instance variable *imMatrix* is computed: **newValidIMM**

5.3.5 The *DuplocPresentationModelProtocolTransformer* class

The *DuplocPresentationModelProtocolTransformer* class extends the *RawMatrix* class update protocol in following way:
If **updatErepairNow** is received, then

1. **changeDrepairNow** is notified to its dependants.
2. the message **updatEadaptModel** is sent to the *transformedProtocolReceiver* instance variable.
   This message must force the *DuplocPresentationModel* instance to adapt to a *RawMatrix* instance change.
3. the message **updatEcorrectViews** is sent to the *transformedProtocolReceiver* instance variable.
   This message must force the *DuplocPresentationModel* instance to force the *DuplocPresentationModelView* instance to update its state and the state of the *DuplocPresentationModelController* instance.

5.3.6 The 'bin value colouring model'

The 'bin value colouring model' defines the grey value used for painting a *bin* dot in the *binplot*, which represents an *Information Mural matrix* element **v** (named *bin value*). Each *Information Mural matrix* element is of type float and has a value in the range [0.0, 1.0]. The most simple method to grey shade each *bin* dot is to have an affine function $\mathbf{f_{colouring}}$, which maps the *bin value* linear on a grey level range between 0.0 and 1.0. In the case of a RGB colouring scheme the three colours are set with the same value:

$$\mathbf{f_{colouring} : [0.0, 1.0\ ] \rightarrow [0.0, 1.0]}$$

[17] e.g. see the attribute *us.rsm.min* in section *5.3.1.4 Attribute value definitions*
[18] see section *5.3.2 RawMatrix class update protocol to its dependants*

$\mathbf{f_{colouring}(x) := a \cdot x + b}$

$\mathbf{f_{colouring}(0.0) := 1.0 = R = G = B \equiv}$ **"white colour"**
$\mathbf{f_{colouring}(1.0) := 0.0 = R = G = B \equiv}$ **"black colour"**

Some preliminary calculation showed the necessity to control this mapping more precisely: The range of match density will normally be below 0.1. This will especially be the case for large *raw matrix* – e.g. a size above 40'000 x 40'000 would mean (as mentioned[19] above) to have a *bin* of 200x200 with a maximal number of 40'000 matches. So, obviously this will be rarely the case. So, the limit of 0.1 described above is already very high (4000 matches).
A more important reason is also to have a possibility to display a fraction of the available range and so to isolate interesting sections of the *raw matrix*.
Therefore a model of a mapping function was defined, which permits to select an interval of the range [0.0, 1.0] :

The '**bin value colouring model**' has following elements:

1) The available *bin values* are defined in the range:
[minAvailableInput, maxAvailableInput] ⊆ [0.0, 1.0]

2) The model has as the definition range:
[minInput, maxInput] ⊆ [minAvailableInput, maxAvailableInput]
with:
minAvailableInput ≤ minInput < maxInput ≤ maxAvailableInput
|maxInput - minInput| ≥ SmallestMaxMinDifference

3) The '**input affine function**' **i** maps values in the range:
i : [minInput, maxInput] → [0.0,1.0]
i(x) := a · x + b
so that:
i(minInput) = 0.0
i(maxInput) = 1.0.

4) The '**selected input affine function**' **g** maps values in the range:
g : [minSelectedInput, maxSelectedInput] → [0.0,1.0]
g(x) := a · x + b
with **[minSelectedInput, maxSelectedInput] ⊆ [0.0, 1.0]**
so that:
g(minSelectedInput) = 0.0
g(maxSelectedInput) = 1.0.

5) The '**monom function**' **m** maps values in the range:
m : [0.0, 1.0] → [0.0,1.0]
$m(x) := 1.0 \cdot x^n$
with **n** ∈ **[1/maximalMonomDegree, maximalMonomDegree**]

6) The '**selected output affine function**' **h** maps values in the range:
h: [minSelectedOutput, maxSelectedOutput] → [minGreyLevel, maxGreyLevel]
h(x) := a · x + b
with **[minSelectedOutput, maxSelectedOutput] ⊆ [0.0, 1.0]**
and **[minGreyLevel, maxGreyLevel] ⊆ [0.0, 1.0]**
so that:
if '**inverted grey level mapping**' selected
then h(minSelectedOutput) = maxGreyLevel, h(maxSelectedOutput) = minGreyLevel
else h(minSelectedOutput) = minGreyLevel, h(maxSelectedOutput) = maxGreyLevel

The new function $\mathbf{f_{colouring}}$ has following definition:
$\mathbf{f_{colouring}}$ **: [0.0, 1.0] → [0.0, 1.0]**

$\mathbf{f_{colouring}(x) := 0.0}$ **, x ∉ [minInput, maxInput]**
$\mathbf{f_{colouring}(x) := (h \circ m \circ g \circ i) (x)}$ **∈ [minGreyLevel, maxGreyLevel]**
, x ∈ [minInput, maxInput]

[19] see section *5.3.1 Representing a raw matrix*

The depiction of an *InformationMural matrix* with the RGB colouring scheme, means that for any matrix element **v** the correspondent dot in the diagram is coloured with:

$$\mathbf{R := G := B := f_{colouring}(v)}$$

5.3.7 Duploc source code information

5.3.7.1 Where is the source code of this project located?

The source code of this project is placed in the category: ***Duploc_InformationMural***.
The source code about the *RawMatrix* class is placed in the category: ***Duploc_RawMatrix***

5.3.7.2 What are the specific classes of this project?

This is the alphabetically sorted list of the 59 classes specific to the project placed in the category ***Duploc_InformationMural***:

- AbstractInformationMuralMatrix
- AbstractMatrixCursorPosition
- AbstractOverView
- AbstractPresentationModelControllerState
- AbstractPresentationModelViewState
- AbstractRawSubMatrix
- AbstractSuperOverView
- AbstractUserSelection
- AffineFunction
- BinValueColorerInterface
- BinValueColorerView
- BinValueColoringModel
- CachedObservationData
- DuplocPresentationModel
- DuplocPresentationModelController
- DuplocPresentationModelProtocolTransformer
- DuplocPresentationModelView
- ForwardingObject
- IdentityFunction
- InformationMuralMatrixCursorPosition
- MonomFunction
- ObservationOnRawSubMatrix
- OverView
- PMCS
- PMCSdummyMode
- PMCSmovingMode
- PMCSnormalMode
- PMCSOverView3LMovingMode
- PMCSOverView3LMovingSelfMode
- PMCSOverView3LNormalMode
- PMCSOverView3LPositioningMode
- PMCSOverView3LPositioningSelfMode
- PMCSOverView3LSpyingMode
- PMCSOverViewMovingMode
- PMCSOverViewNormalMode
- PMCSOverViewPositioningMode
- PMCSOverViewSpyingMode
- PMCSpositioningMode
- PMCSspyingMode
- PMCSSuperOverViewMovingMode
- PMCSSuperOverViewNormalMode
- PMCSSuperOverViewPositioningMode
- PMCSSuperOverViewSpyingMode
- PMCSUserSelectionMovingMode
- PMCSUserSelectionNormalMode
- PMCSUserSelectionPositioningMode
- PMVSInformationMuralMode
- PMVSOverView3LMovingMode
- PMVSOverView3LNormalMode
- PMVSOverView3LSpyingMode
- PMVSOverViewNormalMode
- PMVSOverViewSpyingMode
- PMVSSuperOverViewNormalMode
- PMVSSuperOverViewSpyingMode
- PMVSUserSelectionMode
- ProtocolTransformer
- RawSubMatrixCursorPosition
- SuperOverView
- UserSelection

5.3.8 Implementation concepts

5.3.8.1 Introduction

This section discusses some simple implementation strategies, which were used to keep the source code flexible.

5.3.8.2 The concept of 'self neighborInstance'

One of the problem during the development of an application is the uncertainty of the application topology – this means how the object instances are connected together.

If an instance **A** sends a message $\mathbf{msg_{b1}}$ to an instance **B**, then this instance **A** might have an instance variable **toB**, which references this instance **B**. So, in the source code of **A** each time a message is sent to **B**, there will be something like '… **toB** $\mathbf{msg_{b1}}$ …'.

If this instance **A** should not reference directly the instance **B**, because there is an intermediate instance **S**, which selects a current instance $\mathbf{B_i}$ out of a set, then all messages sent from **A** to **B** should be modified like follows: '… **toS currentB** $\mathbf{msg_{b1}}$… '.

Instead of using an instance variable **toB** to reference a specific neighbouring instance **B**, it is better that the instance **A** sends to itself a message '…**self neighbouringB** …', which returns a reference to the requested neighbouring instance **B**. So, if a change occurs in the topology, then only this method '**neighbouringB'** must be adapted.

This concept was used throughout all defined classes in this project.

5.3.8.3 The concept of 'self topology'

The concept '**self neighborInstance**' can be pushed further: It might be useful to separate object knowledge with application topology knowledge. Therefore if an instance **A** must send a message $\mathbf{msg_{b1}}$ to an instance **B**, then it can send to itself a message '…**self topology**…', which returns the instance having the topology knowledge, and then send to this instance the message '**currentB**': The complete sequence to send a message $\mathbf{msg_{b1}}$ from an instance **A** to an instance **B** would be: ' … **self topology currentB** $\mathbf{msg_{b1}}$ … '.

5.3.9 Used graphical notation

Diagram 5 shows the Diagram A, B and C. The aimed notation for a class diagram was UML. But, difficulties to express the '**self neighborInstance**' concept forced me, to introduce some modifications:

The class diagrams drawn in Diagram A and Diagram B are equivalent. Diagram B is probably more UML conform, but Diagram A expresses better the needs – if a class 'sees' another class instance through a *referencing* method, then this *referencing method* is denoted on the correspondent relation by adding the opening and closing brackets. An instance variable, which references a neighboring instance, is also denoted on the correspondent relation. The visibility symbols of UML were used, but because each method in a Smalltalk class has public visibility and each instance variable has protected visibility some simplifications were made (see Diagram C) – an omitted visibility symbol for a method or a *referencing method* means public visibility; an omitted visibility symbol for an instance variable means protected visibility.

The underlined instances and methods belong to the correspondent class.

Diagram 5. The used class diagram notation

Diagram A) The defined notation

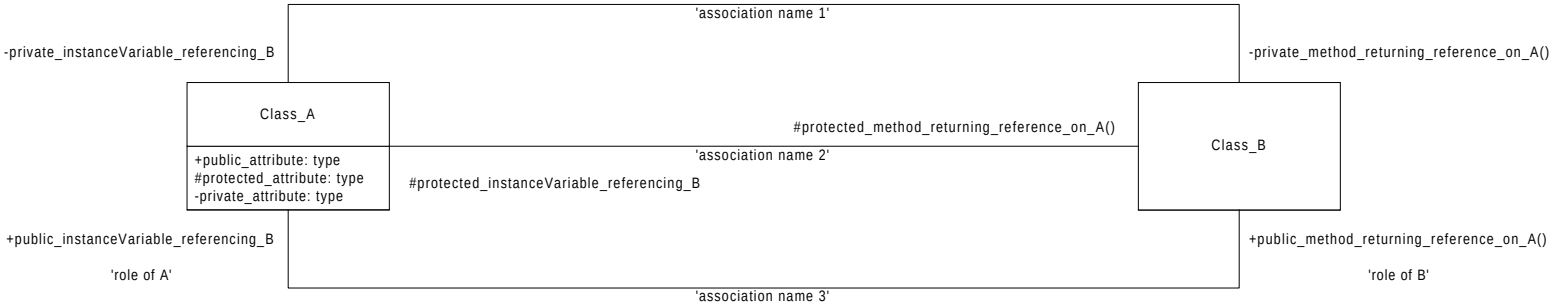


Diagram B) The equivalent notation in UML (?)

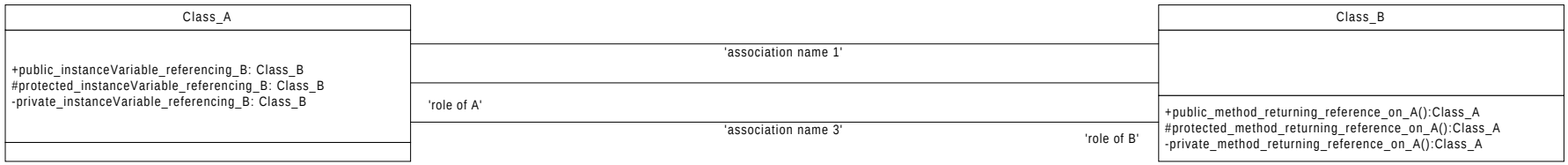
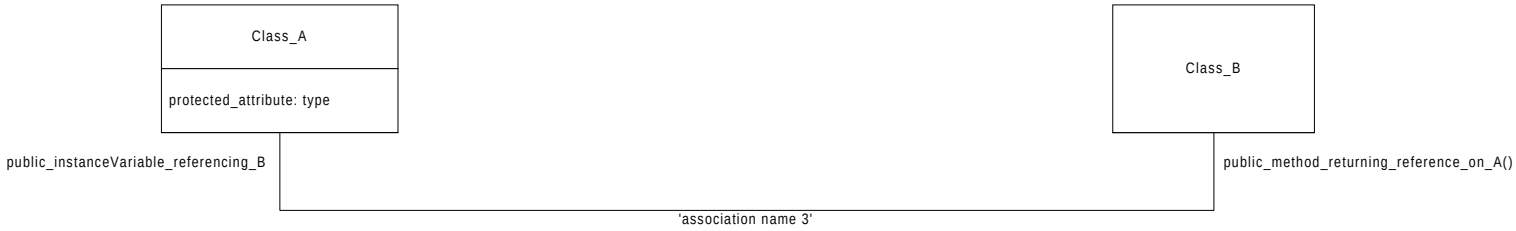


Diagram C) A simplification of Diagram A - omitted visibility symbol means per default ...



6 Conclusions

This chapter summaries the conclusions drawn by the author after the project end. During the development of the new *Graphical User Interface* several difficulties emerged, which induced to formulate following vague development guideline for similar projects. The development of any application should contain two steps. The first step is the description of an appropriated *application concept*, which is based on the original requirement paper. The second step is the description of the *application design*.

Application concept

The definition of an *application concept* can be part of the requirements refinement phase or can be part of the analysis phase. In both case an *application concept* should be verified with the implementation of some (rough) prototype. An *application concept* should consist of three parts: A *system concept* part, a *GUI concept* part and a *topology concept* part.

System concept part

Any implemented solution is embedded in a running environment. In the case of the *Duploc* application, this environment is the VisualWorks 2.5/3.0 framework. The *system concept* part should describe a solution for fulfilling the original requirements without taking in account the running environment. How this description has to occur is difficult to say, but the contents of chapter 2.1, 2.2, 3.2 and 4.1.1 in this document could have been part of the *system concept* of this project.
In this project, it was first necessary to sketch the original *system concept*, which represents the original *Duploc* version. The chapter 2.1 and 2.2 summaries the essence of the found original *system concept*. The next step was to extend this concept. The introduction of the *raw matrix set* had two consequences:
It provides the new *Graphical User Interface* with an interface to the existing application part.
It separates the new *Graphical User Interface* from the access management to the internal *comparison matrix repository*. A new *Duploc* version might provide to the user the functionality's to compose the *raw matrixes* available in the *raw matrix set* by specifying the list of files, which form each *raw matrix*.
The next problem was to define data-structures suitable for being graphical displayed. This problem resulted in the introduction of the *two level* and *three level view representation* of a large *raw matrix* – see section 3.2 and 4.1.1.

The *system concept* part should also describe, what sort of functionality's the application should provide to the user. e.g. moving the *user selection region* inside the current *raw matrix*.

The *system concept* part can contain class or instance diagrams as long as they are not dependant on the running environment.

GUI concept part

The *GUI concept* part should describe the appearance of the application and how the user interacts with the new application. This part is dependant on the chosen running environment. Therefore it involves to implement several small test applications for understanding the possibilities of the environment.
The new *Duploc* application has a simple *GUI concept*: A single view area must present the *two level* and *three level view representation*. This resulted in the introduction in chapter 4.2.4 of the three display modes.

Topology concept part

The *topology concept* part integrates the *system concept* part with the *GUI concept* part by sketching the application topology. e.g. The Diagram 1 shows the *Duploc* topology concept, where the *graphical cloud*, which contains the *'view' and 'controller' state classes*, interact via the *DuplocPresentationModel* class with the *model cloud*, which contains the classes representing the *raw matrix*.

Application design

The application design provides an object oriented solution for the formulated application concept.
A flexible development is aimed, so that important source code is written once and once only.
This is achieved by breaking the application in different components, each one composed by two parts: A part, which is preserved from the actual application topology, and a part specific to the application topology. The *Duploc* example is the *AbstractRawSubMatrix* class hierarchy. e.g. The *UserSelection* class, which is a subclass of *AbstractUserSelection*, is specific to this application topology, but the *AbstractUserSelection* class is not. This topology independence is achieved through the used implementation concept.

7 Appendix

7.1 References

[1] Jonathan Helfman. Dotplot Patterns – A Literal Look at Pattern Languages.
AT&T Research, Murray Hill, NJ 07974, jon@research.att.com

[2] Matthias Rieger: Duploc Tutorial, Version 2.0 Release 1.
Software Composition Group, University of Bern, March 1999.

[JS96] Dean F. Jerding and John T. Stasko. The Information Mural: Increasing Information Bandwidth in Visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, October 1996.