JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik (Systemsoftware)

The S.I.D. Text Model Implementation

Stefan Chiettini

Report 14
April 1999

Author's address:

Institut für Praktische Informatik (Systemsoftware)
Johannes Kepler Universität Linz
A - 4040 Linz / Auhof

e-mail:
stefan.chiettini@ssw.uni-linz.ac.at

JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik (Systemsoftware)

The S.I.D. Text Model Implementation

Stefan Chiettini

Report 14
April 1999

Contents

1. General Structure of SID-Texts

SID-Hypertext is attributed and hyperlinkable text. It is an implementation of the Model-View-Controller (MVC) paradigm, with a clear distinction between the data (text and attribute information held in one single model) and multiple views on the data. In the SID-Implementation, like in most other MVC-Implementations, View and Controller are unified in one class.

In general a SID-Text consists of two parts: A *StringBuffer* that holds the ASCII-Text and a vector holding the corresponding text attributes using so-called 'text runs' (see figure 1). These two parts are encapsulated in the class *Text*. The combination of a text run, and the corresponding ASCII-Text is built by the view and results in the display of an attributed text with different colors, fonts and styles.

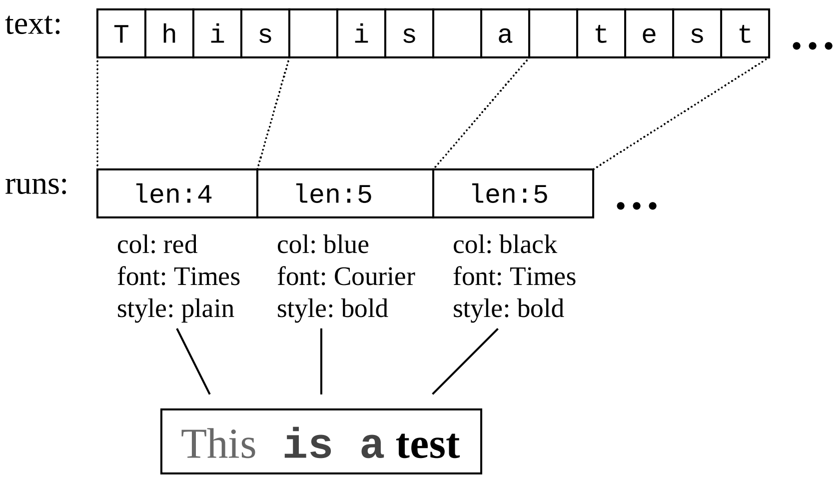


Figure 1: General Structure of a SID Text

A SID text is a subclass of *Viewable*, which is the base class of all models. Each model may have several attached views. Figure 2 shows the two relevant base classes for the SID-Text: *View* and *TextView*. The subclass *ScrolledTextView* is a concrete implementation for a hypertext view (in this case a window with scrollbars and a toolbar for text insertion and editing).

Changes in the model (the text) are propagated by the base class *Viewable* to all views (*TextMessage* objects with the information about the change are sent via broadcast to all views which are themselves responsible to react in an appropriate way).

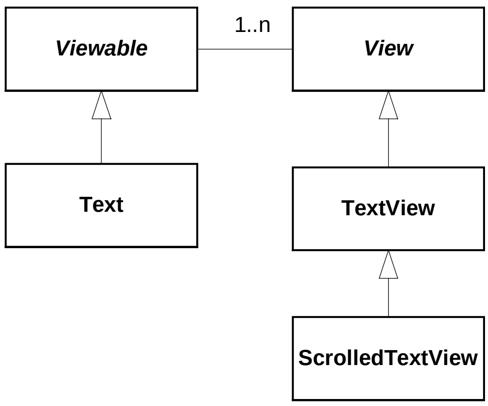


Figure 2: MVC structure

SID-Texts may contain so-called 'text elements'. These are special elements with a graphical representation, that are floating inside the text and cover a rectangular area. Examples for such elements are pictures, URLs or Video elements (called *MediaElems*). Text elements support reaction on mouse clicks (starting any action like showing a video or displaying the element's property sheet). Text elements are represented by special characters in the ASCII text and a special *TextRun* the *TextRun* vector (for more information on elements see chapters 2.1.3 and 3).

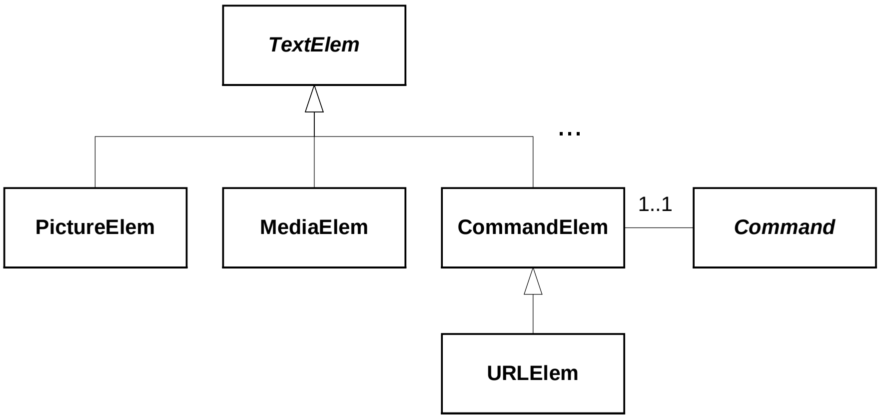


Figure 3: Textelement hierarchy with several examples

Elements have a common base class (see figure 3). SID texts only need to know about the general interface defined in the base class *TextElem*.

2. Detailed Descriptions

This chapter contains detailed descriptions of the implementation and interfaces of the classes involved in the text model. The overall structure is discussed and important methods and parameters are explained.

2.1. TextRun

A *TextRun* represents a piece of text with length > 0 and a set of attributes. The supported text attributes are:

- font name
- font size
- color
- style (plain, bold, italic, italic + bold)

All the characters in the piece of text represented by the *TextRun* share the same attributes.

2.1.1. Public Attributes of class *TextRun*

int *len*	Length of the textpiece. The sum of all lengths is the length of the ASCII text.
Color *col*	Color of the textpiece.
Font *font*	Font information (contains name, size and style attributes)
char *type*	Type of the textpiece, see chapter 2.1.2.
TextElem *e*	Anchor for a text element, see chapter 2.1.3.
short *w, h*	Actual size of the rectangular area covered by the text run.

Table 1: Public *TextRun* Attributes

All attributes shown in table 1 are declared as public, *TextRun* objects serve as simple data records for the text attribute information.

2.1.2. Types of TextRuns

Since there are different types of characters in a text which must be interpreted differently by the views, these types are reflected in the TextRuns (this saves time when iterating over a piece of text). Different types of characters are:

`TEXT`	Normal text characters, they are displayed with the given attributes.
`TAB`	Exactly 1 TAB character is in the ASCII text. The width is calculated depending on preferences in the view that displays the text. If a run has type `TAB`, its length must be set to 1.
`LF`	A line feed character (`'\n'`) is in the ASCII text. This information is kept in the TextRun class for better performance since searching the end of a line is faster when iterating over the TextRun-Vector. Searching the end of a line in the plain text requires checking all characters of a line. In the TextRun-Vector only blocks of characters have to be checked for the `LF` type attribute. If a run has type `LF`, its length must be set to 1.
`EOT`	The end of the text is marked by this type attribute. Only one run with this attribute can be present in a text (of course at the end of the text).
`ELEM`	If a text element resides on the corresponding position in the text, a special character is located in the ASCII-Text and the corresponding TextRun is of this type (see chapter 2.1.3.). Attribute *e* (see table 1) contains a reference to the text element. `len` must be 1.
`UNDEF`	Should never be set in a correct text. This is a possibility to mark errors in a corrupt text.

Table 2: *TextRun* Types

The constants (public static final) shown in table 2 should be used by the views to determine the type of the TextRun that is currently used.

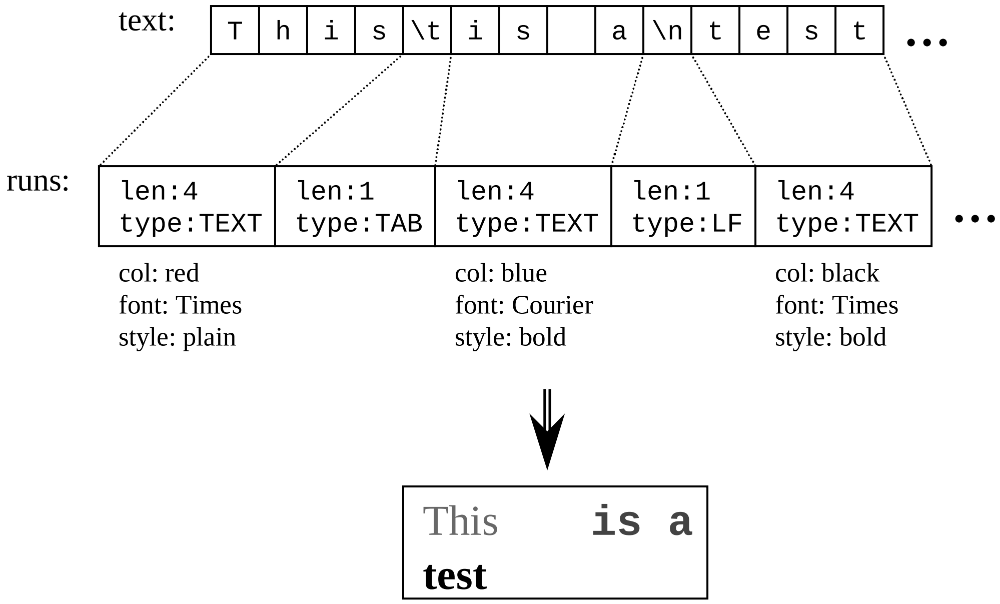


Figure 4: Example with special characters inserted

In figure 4 `'\t'` stands for a tabulator character, the width of the blank space depends on the settings in the displaying view. `'\n'` is the line feed, this information is kept in the TextRun for performance purposes but leads to an increased memory usage since more TextRun blocks are needed to represent a piece of text with the same attributes but multiple lines.

2.1.3. Anchoring Text Elements

Text elements cover a rectangular area, they can be aligned to the top, bottom, center or baseline of the text line they are residing in. They are represented in the ASCII text by a special character (value: `0x1c`) defined in the class *TextElem* (see chapter 3.1.1.) which is the same as the constant `ELEM` shown in table 2. For a text element a single TextRun is reserved with type `ELEM` and a reference to the specific element in the attribute *e*.

When the text is displayed by a view and a text element is found, first its *prep* method is called with the color, font and graphic context of the environment passed as parameters. In this method the element may change its shape, layout or even size in response to the call. The view has to react on changes made by the user. The element is drawn by calling its *paint* method (More information on elements see chapter 3).

Example:

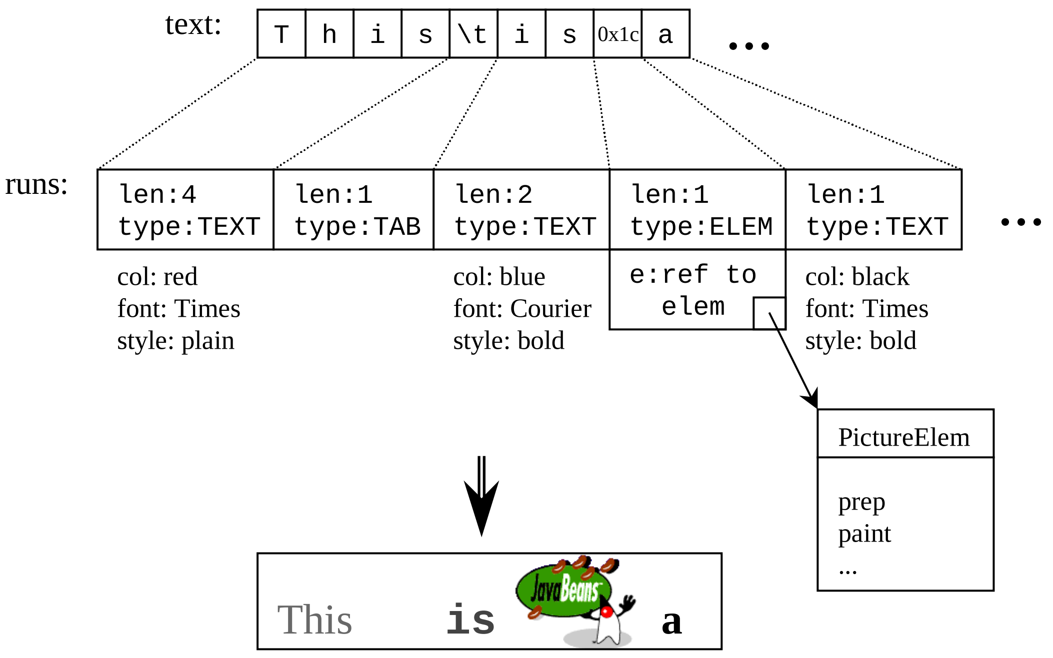


Figure 5: Text Element Anchoring in a SID Text

The example in figure 5 shows a text with an inserted *PictureElem* which is aligned at the bottom of the line. The TextRun with type `ELEM` contains a reference to the text element.

2.1.4. Size Caching

The TextRun caches its actual size in the attributes `w` and `h`. Each time the text is changed, all runs which attributes are affected have to be updated by calling method *updateSize*:

```
updateSize(char[] text, int pos, Graphics g)
```

updateSize recalculates the dimensions of the text block represented by a certain TextRun. *text* is the array with the ASCII text, *pos* is the starting position of the text run, *g* is the graphic context where the characters will are displayed. The size caching must be kept consistent by the *Text* class, additional memory is consumed by the cache, but performance increases significantly when calculating line dimensions (which is always necessary when new a text layout is made).

2.2. TextVector

The TextRun vector named *runs* in the examples in figures 1, 4 and 5 is of class *TextVector*. This is a reimplementation of the class *java.util.Vector*. It contains the same methods as java.util.Vector plus some additional methods. The difference between the two classes is that java.util.Vector was designed for multi purpose collections holding any kind of objects. TextVector is an optimized vector only for TextRuns (it encapsulates an array of TextRuns) which has separate non-synchronized versions of various methods.

2.2.1. Reimplemented Methods

java.util.Vector	**TextVector synchronized**	**TextVector non synchronized**
addElement (Object obj)	*addElement* (TextRun obj)	*addElem* (TextRun obj)
insertElementAt (Object obj, int index)	*insertElementAt* (TextRun obj, int index)	*insertElemAt* (TextRun obj, int index)
removeElementAt (int index)	*removeElementAt* (int index)	*removeElemAt* (int index)
removeElement (Object obj)	*removeElement* (TextRun obj)	*removeElem* (TextRun obj)
Object *elementAt* (int index)	TextRun *elementAt* (int index)	TextRun *elemAt* (int index)
Object *firstElement*()	TextRun *firstElement*()	TextRun *firstElem*()
Object *lastElement*()	TextRun *lastElement*()	TextRun *lastElem*()

Table 3: Comparison of Reimplemented Methods of *TextVector*

Table 3 shows that the synchronized methods of *TextVector* are the same as in java.util.Vector except that any occurrence of *'Object'* in java.util.Vector was replaced by *'TextRun'*. The non-synchronized versions have the same interface as their synchronized counterparts except that the name of the method has changed (the word 'Element' was repaced by the word 'Elem'). In the text model all vector operations are done with non-synchronized methods, which requires that no concurrent threads change the same model. *TextVector* was implemented for better performance escpecially in JDK virtual machines since synchronisation and type guards are expensive operations.

2.3. MVC base classes

The implementation of the MVC paradigm in SID depends on the abstract classes *Viewable* and *View*. The model is a subclass of *Viewable*, the view and controller part ist encapsulated in the class *View*.

2.3.1. Viewable

The abstract class *Viewable* is responsible for the management of all attached views. Views are kept in a java.util.Vector and are accessed by the methods

- public void *addView* (View v)
- public void *removeView* (View v)
- public View *getFirstView* ()

Broadcasts are sent by method

- public final void *broadcast* (Message m)

Broadcasted data is encapsulated in a *Message* object, which is sent to all attached views.

2.3.2. Messages

Changes in the model are sent to all views by broadcasting *Message* – objects. All messages are subclasses of the abstract base class *Message*:

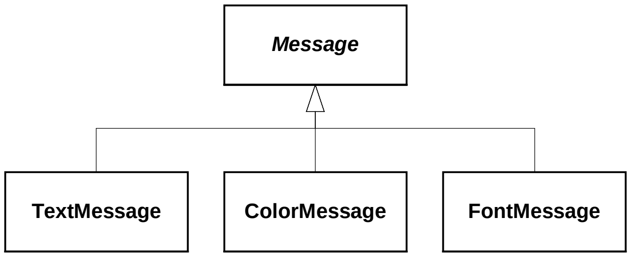


Figure 6: Message hierarchy

Inside the SID text model only *TextMessage* objects are used to inform views of changes in the text. A *TextMessage* contains a reference to the changed model, the action code and the begin and end index of the changed span of text:

- Text t
- int action
- int from, to

Possible action codes for the attribute *action* are provided by four constants defined in *TextMessage*:

Codename	Value	Description
`replace`	0	The text in the span *[from..to[* was replaced.
`delete`	1	The text in the span *[from..to[* was deleted.
`insert`	2	The text in the span *[from..to[* was inserted.
`update`	3	The text in the span *[from..to[* was changed (font, size, color, ..).

Table 4: Text Changes represented by *TextMessage*

The classes *ColorMessage* and *FontMessage* are not used by the text model but by the toolbar of the views to inform the view that some action has to be performed (change the color or font of a certain text span – these changes are performed by corresponding methods).

2.3.3. GraphicContextFactory

Since most classes involved in the text need information for rendering or calculating character dimensions, the graphic context information is needed all over the model. Those classes directly or indirectly derived from *java.awt.Component* already have their own context, but classes like *Text* are not subclassed from any AWT component and therefore need a server (or factory) object to provide the appropriate graphic context when needed. Graphic contexts change during program execution, they cannot be generated at application startup and stored in the model. For this reason the *GraphicContextFactory* interface was created.

It is located in the package SID and offers one single method:

```
package SID;

import java.awt.*;

public interface GraphicContextFactory{
     public Graphics getGraphics();
}
```

Any class implementing this interface must have direct or indirect access to an AWT component providing the graphic context. *Viewable* encapsulates a graphic context factory which can be used by any subclass of *Viewable* (e.g. *Text*).

2.3.4. View

View is the base class for all view/controller parts in the SID MVC implementation. It is an abstract class that already implements several mechanisms and enforces some interfaces to be implemented by the subclasses.

2.3.4.1. Graphical Representation

A View must have a graphical representation and therefore has to implement the *GraphicContextFactoy* interface in order to access the graphic context and forward it to the model.

Graphical output is done in the methods *update* and *paint*. *View* itself is no AWT component but typically encapsulates one (e.g. a *java.awt.Canvas* or *java.awt.Panel*) and delegates display requests to the encapsulated component. This component can be retrieved with method *getDisplayPart* (e.g. to insert it into a frame). Figure 7 shows a typical layout of a text model (it is very similar to the layout used in the SID text model). The view refers to an external part consisting of a Panel that connects a Canvas for the text display and two scrollbars for user navigation.

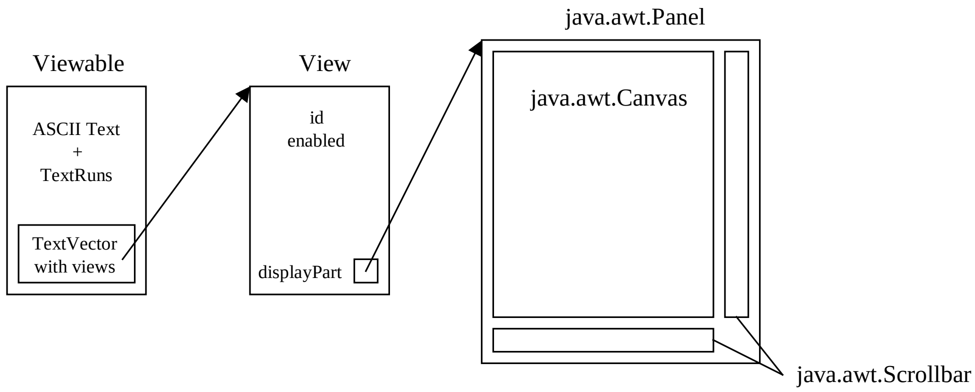


Figure 7: Typical Model-View Combination

A *View* is not an AWT Component and therefore cannot be inserted directly into a window. The external display part (in figure 7 the *java.awt.Panel*) is placed in the GUI and forwards the AWT call of method *paint* to the view which is responsible to paint the model into the passed graphic context.

The Panel is responsible to translate the graphic context due to user scrolling. The Canvas forwards the call of its *paint* method to the view which will actually draw the model on the passed graphic context.

2.3.4.2. Reaction on Events

- Views react on messages sent by the model, this is done in method *handle*.

- *java.awt.event* interfaces that have to be implemented are *KeyListener*, *MouseListener* and *MouseMotionListener*. Even if a view is not interested in some of these events, the interfaces have to be implemented (in the case of an event that is not of interest for the view, the methods are empty and the view is not registered as a listener – this guarantees that no runtime overhead is generated).

- Views may be enabled and disabled (attribute *enable*), the value of this flag is evaluated when broadcasting a message, which is done with the class method *broadcast*, no message objects are sent to disabled views. A View is enabled by default in its standard constructor.

2.3.4.3. Identification / Load & Store Mechanism

Several views may reference the same nodes. To preserving this connected state when storing connected views, a unique identification of the views is needed. This is done with an integer ID number that is managed with the methods *getID* and *setID* (One possible application of the ID is behaviour diagrams, where several views are arranged as boxes and interlinked with arrows. The views form a graph that has to be preserved.).

Models and views are stored on *DataOutputStreams*. Each object is responsible to make the appropriate super calls when loaded, then load itself and eventually tell attached objects to load themselves. The text model is a hierarchical tree structure with no cyclic references that have to be resolved. Other views (e.g. Behaviour Diagrams) than simple text applications need additional information about the structure (e.g. ID of different views) they are storing.

The load & store mechanism makes no use of the Java serialization interface because this interface is slow and generates relatively large files. The optimized code is approximately 2-3 times faster and creates files with a size 50% smaller than a file serialized with the standard Serialization mechanism.

Storing a text results in the following calling chain:

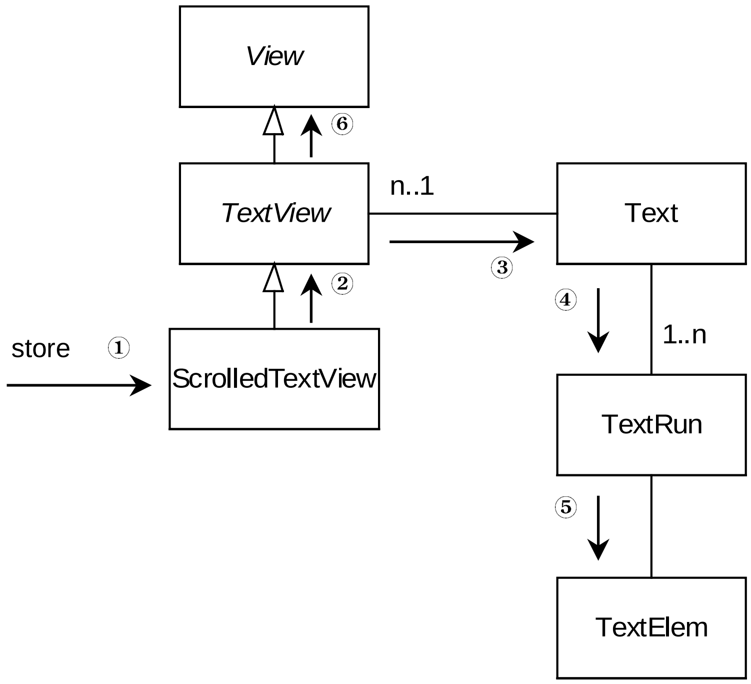


Figure 8: Storing of a SID Text (Example)

Figure 8 shows how a store message is handled by the involved classes:

The user selects a menu item to store a text. The *Frame* class that contains the menu prompts the user for the file name, creates or reopens the selected file and creates a DataOutputStream required by the view classes. This stream is passed to the view class attached to the frame (in this example a *ScrolledTextView*)- ①.

The *ScrolledTextView* eventually stores specific data and makes a super call to its base class *TextView* ②. The *TextView* tells the model (in this example a *Text*) to store itself ③. *Text* first stores the ASCII text, then iterates over its internal TextRun-Vector and tells each TextRun to store itself ④. If the TextRun contains text, it writes the text attributes to the DataOutputStream. If there is a text element attached, the TextRun tells the text element to store itself⑤. When the text is completely stored, *TextView* makes a super call to the base class *View* ⑥, which has to store its internal state (only the ID). Loading the structure works in the same order.

2.4. Class 'Text'

Text is a subclass of *Viewable*. It implements an attributed text with embedded extensible text elements.

2.4.1. Static Fields

String *hypertextTag* = "hTt"	the tag inserted in the header of all text files containing a SID text
int *baseline* = 0	text element alignment
int *bottom* = 1	text element alignment
int *top* = 2	text element alignment
int *tabUnit* = 5	number of pixels per tab count
Font *defaultFont*	default font for an empty text

2.4.2. Public Fields

StringBuffer *text*	ASCII text
TextVector *runs*	TextRuns corresponding to the ASCII text
int *tabs*	tab width (= tabs * tabUnit)
TextRun *defaultRun*	default TextRun for empty texts

2.4.3. Constructors

- `Text()`
  Create an empty text with default color black and default font.
  No *GraphicContextFactory* is attached.
- `Text(GraphicContextFactory gcf)`
  The same as the default constructor with attached *GraphicContextFactory*.
- `Text(String s, GraphicContextFactory gcf)`
  Create a text and insert String *s* as initial value.

Methods *reset* and *setDefault* are used to initialize a newly created text object.

2.4.4. Low Level Methods and Helper Classes

Low level methods are private methods used for internal text manipulations, especially managing the attribute vector while preserving consistency of the length information in the TextRuns.

`private final int split(int pos)`

Initial Situation:

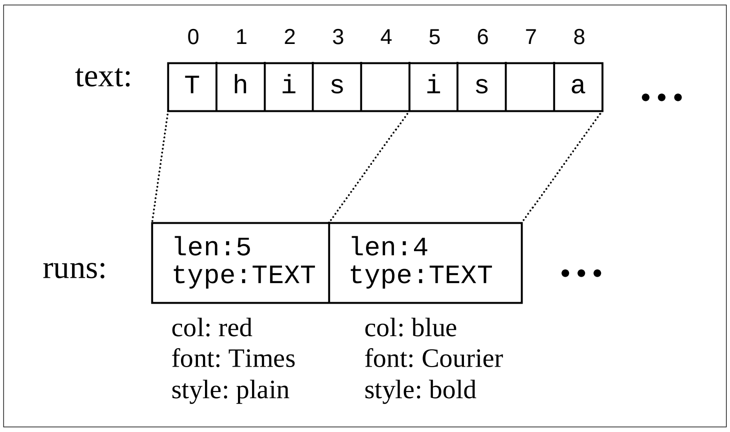


Figure 8: Consecutive TextRuns have different Attributes

Method split receives a text position and splits the TextRun covering this position into two consecutive runs with the same attributes and their length summing up to the length of the splitted TextRun. Position *pos* is held by the new TextRun. If a splitted TextRun has length 1, it is reduced to length 0 and the inserted run gets length 1. Figure 8 shows a possible initial configuration and the result if split(2) is called can be seen in figure 9.

`Split(2)` called:

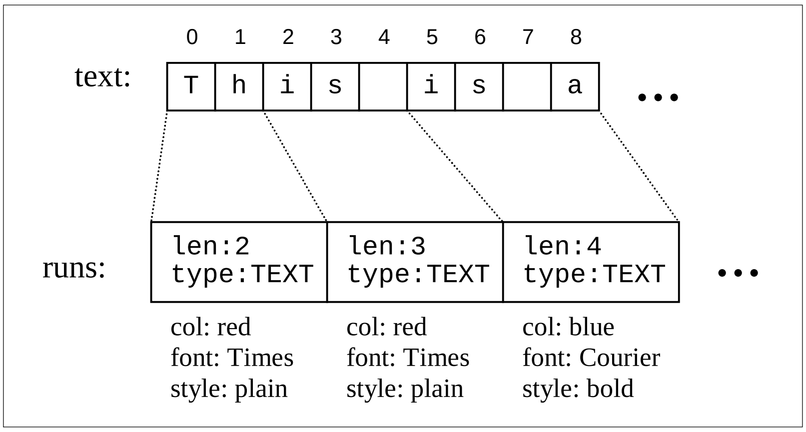


Figure 9: First TextRun is split

`private final void merge(int from, int to)`

Method *merge* receives an index span of size *[from .. to[*. The method merges all consecutive TextRuns with the same attributes in this range of the TextRun-Vector to one TextRun with the given attributes but summed length. merge also removes TextRuns with length <= 0 from the TextRun-Vector. To restore the initial situation shown in figure 8 from the configuration in figure 9, `merge(0,2)` has to be called (any value >= 2 will suffice for the parameter *to*). A public method named *merge()* is provided to merge the complete text.

Usage of *split* and *merge*

All manipulation of the underlying text is done via *split* and *merge*.

Possible change operations are:

- Insertion of plain ASCII Text (or a single character)
- Insertion of attributed Text
- Insertion of special characters (LF, TAB)
- Insertion of a text element
- Changing of attributes in a text range
- Removing a text range

These are the basic operations that can be combined to form more complex text operations. All insertion operations need the position in the text where the insertion has to take place. Change- and remove-operations need the text range that has to be manipulated.

Given the following initial situation all operations are discussed:

The initial situation may be some text with certain attributes. All consecutive TextRuns contain different text attributes.

text	`Hello you funky world!`
runs	`len:22` `type:TEXT`

Inserting any kind of text, characters or elements at a given position results in the splitting of the affected TextRun (e.g. insert at position 9 → `split(9)`). The ASCII text itself is not affected by the split.

text	`Hello you funky world!`	
runs:	`len:9`	`len:13`
text pos:	0-8	9-21

New TextRuns are inserted in the gap between the two resulting blocks, the ASCII text is manipulated directly.

e.g. Insert character `','` at position 9

text:	`Hello you, funky world!`		
runs	`len:9`	`len:1`	`len:13`
text pos:	0-8	9-9	10-22

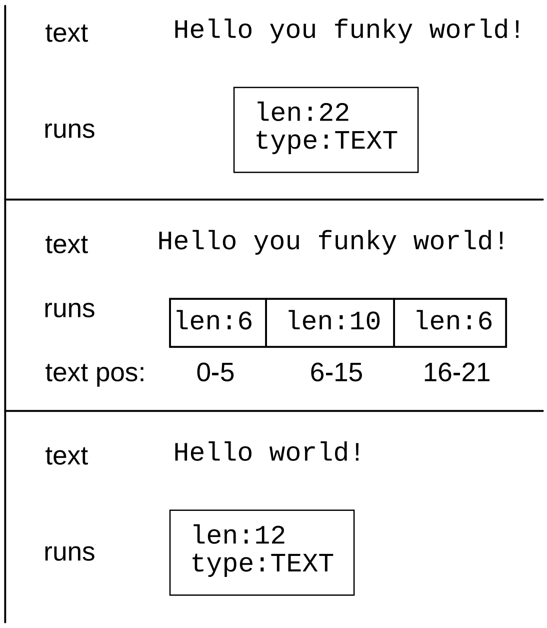
After the changes are made, the text model tries to merge the changed TextRuns and build blocks as long as possible. In the example this leads to one single TextRun with a length of 23 characters. When special characters or elements are inserted, merges have no effect.

text	`Hello you, funky world!`
runs	`len:23` `type:TEXT`

Removing or changing parts of the text requires to specify the text range that is affected. This results in two *split* operations isolating the TextRun(s) that have to be removed (or assigned different text attributes).

e.g. remove the text at span *[6..16[*
→ `split(6) + split(16)`

The TextRun at position 6-15 may be removed or changed and the ASCII text has to be manipulated according to the desired operation. A final *merge* removes redundant TextRuns.



`void` ***`copyAttribs`*** `(String s, int idx, int pos, TextRun r)`

copyAttribs is used internally in combination with *split* and *merge* to break inserted ASCII text into corresponding TextRuns (normal text, line feeds, tabulators). The string *s* is inserted at text position *pos*, the generated TextRuns are inserted at position *idx*. Parameter *r* holds the attributes of the preceding TextRun – these text attributes are assigned to the inserted string.

`class` ***`TextPosition`***

A logical position in the SID text model consists of the character position in the ASCII text and the index of the corresponding TextRun. Provided this information, fast manipulations and investigations are possible. *TextPosition* objects serve as data records, they contain three constructors to build *TextPositions* and export two attributes representing the (ASCII-)text position (attribute *textPos*) and the index in the TextRun vector (attribute *attrIdx*). The default *TextPosition(0,0)* represents the beginning of a text (first character, first TextRun).

`class` ***`LineProperty`***

The properties of lines of text are represented by this class. The information is used by the views to align the lines according to their size and the text model to calculate text positions out of screen coordinates. *LineProperty* contains the dimension of the rectangle surrounding the text line and the maximum descent. The width of a text line is the sum of the widths of the text blocks in the line. The height of a line is the height of the highest text block (which consists of the ascent and descent). The maximum descent is needed to calculate the offset of the baseline for the blocks with smaller descents than the maximum (which defines the distance from the bottom of the line to the baseline).

Attributes of class *LineProperty* (all values in pixels)

`int height`	Height of a piece of text (inherited from *Dimension*).
`int width`	Width of a piece of text (inherited from *Dimension*).
`int maxDesc`	Maximum descent of the line.

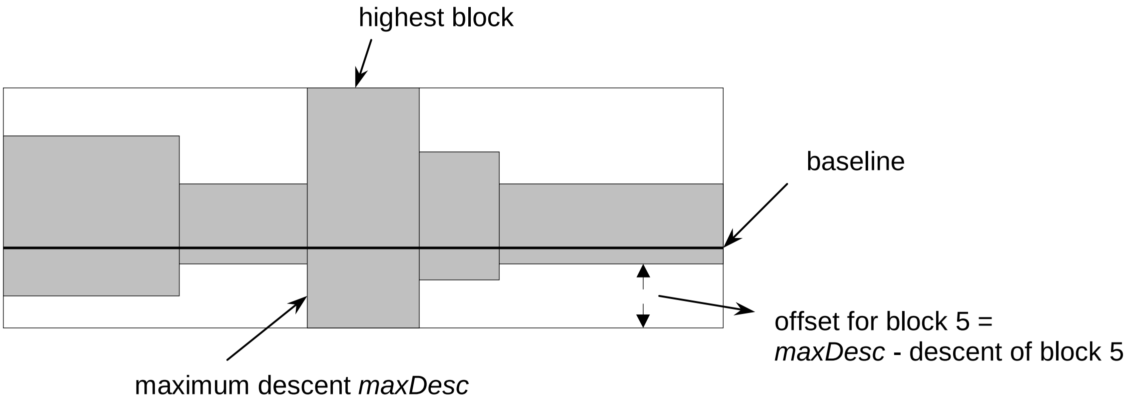


Figure 10: Line Property Description

Figure 9 shows an example for a text line consisting of five blocks with different attributes and lengths. Block 3 is the highest block and has the maximum descent that defines the position of the baseline. The other blocks have to be moved towards the top of the line when displayed in order to form a common baseline. The width of the line is the sum of the widths of the five blocks.

2.4.5. Accessing Text

char *charAt*(int pos)	Retrieve character at text position pos.
char[] *getChars*(int from, int to)	Retrieve characters in text span *[from, to[* as character array.
String *getText*(int from, int to)	Retrieve characters in text span *[from, to[* as String.
TextRun *getRunAt*(int pos, TextPosition start)	Retrieve the TextRun which contains the attributes of the character at text position *pos*, start searching at TextPosition *start*.

2.4.6. Manipulating Text

void *insertText*(String s, int pos)	Insert String *s* at text position *pos*.
void *insertText*(Text t, int pos)	Insert Text *t* at text position *pos*.
void *appendText*(String s)	Append String *s* at the end of the text.
void *concat*(Text t)	Append Text *t* at the end of the text.
void *deleteText*(int from, int to)	Delete text span *[from, to[*.
void *insertTab*(int pos, TextPosition start)	Insert tabulator character at text position *pos*, start searching at text position *start* (increases performance of autoindent).
void *insertChar*(char ch, int pos, TextPosition start)	Insert character *ch* at text position *pos*, start searching at text position *start* (increases performance of editing).
Text *cut*(int from, int to)	Cut out text span *[from, to[* and return as Text.
Text *copy*(int from, int to)	Copy text span *[from, to[* and return as Text.

2.4.7. Calculating Size

Dimension *size*(Graphics g)	Calculate size of the entire text as it will be displayed in the graphic context *g*.
int *getWidth*(TextPosition pos, Graphics g)	Calculate the width of the item at text position *pos* in graphic context *g*: character: width of character tabulator: width of tab (depends on tab sizes) text element: width of text element `lf, eot`: 0

2.4.8. Handling Text Lines

int *getLineLength*(TextPosition pos)	Get number of characters in the line (LF is not counted).
int *lines*()	Get number of lines in the text.
TextPosition *getLineNum*(int num)	Get num'th line, Return text position of first character in the line. Line count starts at 0.
TextPosition *getLine*(int y, Graphics g)	Get line that contains the given y-coordinate. *g* is needed for line property calculations.
LineProperty *GetLineProperty*(TextPosition beg, Graphics g)	Get properties of line starting at text position *beg*.
TextPosition *getLineBeg*(TextPosition pos, Graphics g)	Get the TextPosition of first character in the line containing text position *pos*.
void *getLineBeg*(TextPosition pos, TextPosition beg, Graphics g)	Get TextPosition of first character in the line containing text position *pos*. Return the result in the output parameter *pos* for better performance.
TextPosition *getLineEnd*(TextPosition pos, Graphics g)	Get TextPosition of last character in the line containing text position *pos*.
boolean *getNextLine*(TextPosition pos)	Get the line after the line containing text position *pos*.
boolean *getPrevLine*(TextPosition pos)	Get the line before the line containing text position *pos*.

2.4.9. Character Positions

TextPosition *getCharPos*(int pos, Graphics g)	Get the TextPosition (+ TextRun information) of the character at text position *pos*.
Point *getCharLocation*(TextPosition pos, int mode, Graphics g)	Get the coordinates of the lower left corner of the rectangle surrounding the character at text position *pos*. Alignment modes: *baseline, bottom, top*
void *getCharPos*(int x, TextPosition line, Graphics g)	Get text position of character at x-coordinate in the text line. *line* must be the first character in the line. The position is returned in the parameter *line*.

2.4.10. Changing Attributes

void *setColor*(int from, int to, Color c)	Set color of text span *[from, to[* to color *c*.
void *setStyle*(int from, int to, int style)	Set style of text span *[from, to[* to given *style* (may be Font.PLAIN, Font.BOLD, Font.ITALIC)
void *setSize*(int from, int to, int size)	Set size of text span *[from, to[* to given *size* in pixels.
void *setFont*(int from, int to, String name)	Set font of text span *[from, to[* to font *name*.
void *setFont*(int from, int to, Font f)	Set font of text span *[from, to[* to font *f*.

2.4.11. Serialization

void *load*(DataInputStream f)	Load a text from input stream *f*.
void *loadSecureAscii*(DataInputStream f)	Load only ASCII information from input stream *f*.
void *store*(DataOutputStream f)	Store a text to output stream *f*

2.4.12. Handling Text Elements

void *insertElem*(TextElem e, int pos, TextPosition start)	Insert text element *e* at text position *pos*, start searching at text position *start*.
TextElem *elemAt*(int pos)	Get text element at text position *pos*. Return null if no text element is present at text position *pos*
int *elemPos*(TextElem elem)	Get text position of text element elem.

2.4.13. Text Element Iterator

The iterator is active and not robust against changes in the underlying text while iterating over text elements.

private int *ElemPos*	Current position of the element iterator.
TextElem *firstElem*()	Get The first text element in the text.
TextElem *nextElem*()	Subsequent calls return the next element until the end of the text is reached.

3. Text Elements

Text elements are generic elements that float in the text and support graphical representation and user interaction. They are located in the package *SID.Hypertext.TextElems*. A text element must support the following functions:

- Draw itself in its rectangular area.
- Offer a property dialog to change data contained in the element.
- Load & Store itself inside the text.
- React on user input (mouse clicks and/or keyboard).

3.1. Base Class – TextElem

TextElem is the abstract base class of all text elements. It defines the minimal common interface for all subclasses.

3.1.1. Static Fields

static final char *CHAR* = 0x1c	Filler character inserted into the ASCII text to represent the text element.
static final int *bottom* = 0	Alignment constant
static final int *center* = 1	Alignment constant
static final int *top* = 2	Alignment constant
static final int *baseline* = 3	Alignment constant

3.1.2. Public Fields

public int *w, h*	Dimension of the text element
protected int *x, y*	Coordinates of the lower left corner of the surrounding rectangle.
public Text *base*	Link to the hosting text
public int *align*	Alignment mode
public Color *backGrnd*	Background color

3.1.3. Methods

construction	
public *TextElem*()	Default constructor (size is 0 x 0)
public *TextElem*(int w, int h)	Constructor to initialize attributes, provided for subclass supercalls.
abstract boolean *init* (Text t, int textPos, Frame parent)	Should be called by the creator (e.g. a view or controller) after a new text element was created (initializes the element, e.g. prompts the user for certain informations). Return true if the initialization was successful, false if an error occurred or the user canceled the initialization.
static TextElem *createTextElem* (String s, Text t, DataInputStream f)	Create a TextElement of type *s* in text *t*, loading data from stream *f*. *init* is not called since the object is loaded (not newly created).
painting	
void *clear*(int x, int y, Graphics g)	Fill rectangular area covered by the element with the background color (default: *java.awt.Color.white*).
void *paint*(int x, int y, Graphics g)	Should be overridden and filled with the code that paints the element at the screen position (*x, y*). The method of this superclass sets the attributes (x, y) for internal purposes.
void *update*(int x, int y, Graphics g)	Calls *clear* and *paint.*
abstract void *prep* (Color c, Font f, Graphics g)	Should be overridden, is called before the element is painted. This method contains code that prepares the element to be painted with color *c* and font *f*. The layout and size of the element may be changed in this method.

interaction	
abstract void *activate*(View from)	Is called when activated by the user (e.g. mouse clicks). The parameter contains the view that was the origin of the activation.
abstract void *showProperties* (int x, int y, Frame parent)	Display a properties dialog in parent frame *parent* at screen coordinates (*x*, *y*).
void *handle*(Message m)	React on message *m*.
String *helpMsg*()	Return an explanation string (e.g. to be displayed in a status bar or a tooltip text).
position	
Point *getLocation*()	Get position of the text element.
int *elemPos*()	Get text position of the text element.
serialization	
void *load*(DataInputStream f)	Load element from input stream *f*.
void *store*(DataOutputStream f)	Store element to output stream *f*.

3.1.4. Construction of Text Elements

There are two ways to construct a text element:

Instantiate a new element

```
PictureElem e = new PictureElem();
e.init(text, pos, parent);    // prompt user for image file
```

Additional constructors are available to construct a new *PictureElem*. The combination of default constructor and the call to method *init* is used by the views in SID to create and initialize yet unknown elements.

Load an element from a given file

```
TextElem.createTextElem("PictureElem", text, stream);
```

This call creates a new *PictureElem* using the default constructor and initializes the element with the stored data from the DataInputStream. The position in the stream has to be correct (it is not checked by the method *createTextElem*).

3.1.5. Interaction with the View

All text elements have to support various mechanisms to be integrated into a view.

Drawing of Text Elements

Prior to a call of the paint method, all text elements will be sent a *prep* message. In the method prep, the text element is informed that it is about to be drawn with the given color and font in the given graphic context. Some elements may change their layout, take special calculations or even change their size. The view has to guarantee that the paint method is called with the same parameters as *prep*.

An example for the usage of method *prep* is the text element *DateElem* (for more details see chapter 3.3.1.). This element shows the date of the last store operation of the document it is located in, representing a simple version control mechanism. DateElems are displayed textually in the font of the surrounding text. If the attributes of the surrounding text change, the look and size of an affected DateElem has to change, too.

New text attributes are stored in the corresponding TextRuns including the TextRuns that hold text elements (see chapter 2.1.1.). But the change does not affect the elements themselves. The actual size of elements is calculated and cached in the attributes *w* and *h* each time the element is drawn (see chapter 2.1.4.). So, when the line containing an element is about to be drawn, the correct size of the element is invalid (the cache of the element still contains the old values) and has to be recalculated. The recalculation is done in the method *prep* of the element. When all elements of the line are updated, the correct line dimensions are calculated and the line is drawn.

User Interaction

The user may activate a text element (e.g. clicking the mouse or pressing a hotkey). The view is responsible to resolve the user action and call the text element's *activate* method. This method starts the specific action (e.g. display a picture, show a video, start a thread, etc.). To manipulate certain properties of the element, method *showProperties* is supposed to open a Properties Dialog where the user may change the attributes of the text element.

Method *handle* is provided for future extensions and yet unknown messages eventually needed for inter-element communication. *handle* receives a message record with specific data (see chapter 2.3.2.).

Serialization

Like *TextRuns*, text elements have to fit into the serialization mechanism of the SID-model. Each element is responsible to store itself (and perform a supercall prior or after the store operation). The base class *TextElem* stores the cache and alignment information.

3.2. The Text Element Registry

Text elements are managed by a registry, which combines a specific element with a menu item and a hotkey. The registry is a serialized hashtable, the elements are inserted in a dialog (private class *RegDialog*). The hashtable stores objects of type *ElemData* (containing the classname plus package information, the menu string and the hotkey character). The registry information is stored in the file *'elemreg.dat'* – see static attribute *regFile* of class *TextElemRegistry*.

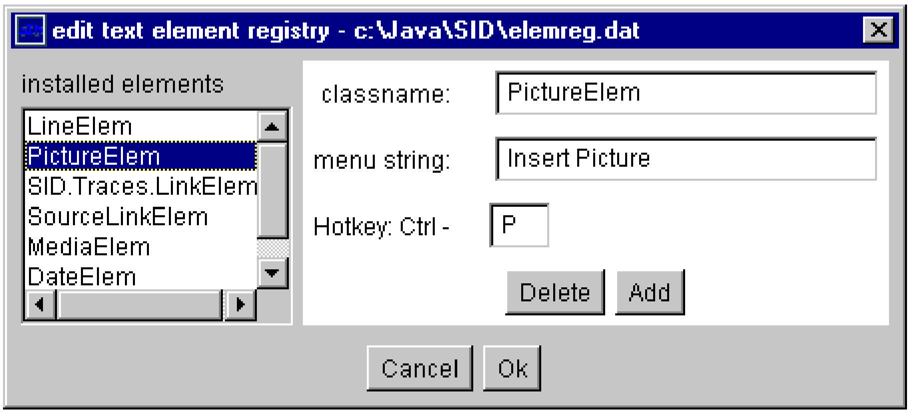


Figure 11: Registry Dialog

Since the registry is an abstract data structure with only one instance, all methods are implemented as static methods.

3.2.1. Static Methods

static void *setInstallDir*(String dir)	Set the path to the registry file 'elemreg.dat'.
static String *getInstallDir*()	Get the path and name of the registry file.
static void *edit*(Frame f)	Open the dialog to manage the installed text elements. Frame *f* is the parent frame for the dialog.
static TextElem *CreateTextElem* (String menuname, Frame f)	Create a new instance of the element class registered with the given menu string. Parameter *f* is provided for error message dialogs.
static Menu *buildMenu*(ActionListener a)	Build a menu named *'elements'* containing all registered elements. ActionListener *a* is registered at the menu (and typically forwards the selected menu string to method *createTextElem*).
static Hashtable *loadReg*()	Load registry from file `'elemreg.dat'`.
static void *storeReg*()	Store registry to file `'elemreg.dat'`.

3.3. Examples

This chapter contains two examples of text elements, *DateElems* (very simple, no user interaction, no properties, varying size depending upon surrounding text) and *PictureElems* (displaying a picture – typically iconified. When clicking on the icon, the picture is displayed full-size in a separate window).

3.3.1. DateElem

A *DateElem* shows the date of the last store operation of the text in which it is embedded. When receiving a store message, it updates its internal data before actually storing itself and broadcasts the change. The element has a textual representation and therefore an alignment to the baseline of the text (see constructor). The element supports no user interaction and has no editable properties (methods *activate* and *showProperties* are empty). The element dimension depends upon the attributes of the surrounding text (see method *prep*), its style may be changed like normal text.

```java
package SID.Hypertext.TextElems;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.text.*;

import SID.Hypertext.*;
import SID.Utils.*;

public class DateElem extends TextElem implements Cloneable{

   public Date date;
   private String s;
   private static final String months[] =
      {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

   public DateElem(){
      super();
      date = new Date(System.currentTimeMillis());
      s = null;
      align = TextElem.baseline;
   }

   private String getString(){
      if (s == null) s = DateFormat.getDateInstance().format(date);
      return s;
   }

   public Object clone() throws CloneNotSupportedException{
      DateElem d = (DateElem)super.clone();
      d.date = new Date(date.getTime());
      return d;
   }

   public boolean init(Text t, int textPos, Frame parent){ return true; }

   public void paint(int x, int y, Graphics g){
      super.paint(x, y, g);
      String s = getString();
      Font f = g.getFont();
      y+=g.getFontMetrics(f).getAscent();

      g.drawString(s, x, y);
      int wid = tool.stringDim(g, s, f).width-2;
      for(int i=0; i < wid; i+=4)
         g.drawLine(x+i, y, x+i+1, y);
   }

   public void activate(View from){ /* do nothing */ }
   public void showProperties(int x, int y, Frame from){ /* do nothing */ }

   public void prep(Color c, Font f, Graphics g){
      Dimension d = tool.stringDim(g, getString(), f);
      w = d.width; h = d.height;
   }

   public void load(DataInputStream f)throws IOException{
      super.load(f);
      date = new Date(f.readLong());
   }

   public void store(DataOutputStream f)throws IOException{
      super.store(f);
      date = new Date(System.currentTimeMillis());
      f.writeLong(date.getTime());
      int pos = elemPos();
      base.broadcast(new TextMessage(base, TextMessage.update, pos, pos+1));
   }
}
```

3.3.2. PictureElem

Pictures are inserted into texts using *PictureElems*. The picture is displayed in a shrinked form, and is displayed full-size when clicked (activated). When a new picture is inserted, a *FileDialog* is displayed in the *init* method which prompts the user for the name of the file being inserted.

3.3.2.1. PicDraw

The private class *PicDraw* is an *ImageObserver* which informs the views to update their graphical representations when the image is loaded successfully.

```
class PicDraw implements ImageObserver{

   PictureElem e;
   Text t;

   PicDraw(PictureElem p, Text t){
      e = p; this.t = t;
   }

   public boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height){
      Toolkit tk = Toolkit.getDefaultToolkit();
      if (infoflags == ALLBITS){
         int pos = e.elemPos();
         t.broadcast(new TextMessage(t, TextMessage.update, pos, pos+1));
         return false;
      }
      return true;
   }
}
```

3.3.2.2. PictureCanvas

The *PictureCanvas* is the surface which displays the picture in the separate window when activated. It provides the graphic context to draw the picture. It is connected with the displaying frame (see constructor). Additionally it implements a *WindowListener* which closes the window when the close button is clicked. The image is stored GIF-encoded inside the text.

```
class PictureCanvas extends Canvas implements WindowListener{
   private Image img;
   private Frame win;

   PictureCanvas(Image i, Frame w){
      super(); img = i; win = w;
   }

   public void paint(Graphics g){ g.drawImage(img, 0, 0, null); }
   public Dimension getPreferredSize(){
      return new Dimension(img.getWidth(null), img.getHeight(null));
   }

   public void windowClosing(WindowEvent e){
      win.setVisible(false);win.dispose();
   }

   public void windowActivated(WindowEvent e){}
   public void windowClosed(WindowEvent e){}
   public void windowDeactivated(WindowEvent e){}
   public void windowDeiconified(WindowEvent e){}
   public void windowIconified(WindowEvent e){}
   public void windowOpened(WindowEvent e){}

}
```

3.3.2.3. PictureElem

PictureElem holds the displayed image, connects the picture with the *ImageObserver* and determines the state of the image (attribute *ok*). PictureElem reacts on user input (mouse click) by implementing method *activate* (action: open a new Frame with a PictureCanvas displaying the image).

```
public class PictureElem extends TextElem implements Cloneable{
   private Image img;
   private ImageObserver pd;
   public boolean ok;

   public static int maxW = 40;
   public static int maxH = 40;

   PictureElem(){
      super(); img = null; ok = false; pd = null;
   }

   public PictureElem(String imageName){
      super();
      Image i = Toolkit.getDefaultToolkit().getImage(imageName);
      initp(i);
   }

   public PictureElem(Image i){
      super(); initp(i);
   }

   private void checkFlags(){
      int flags = 0;
      while (((w == -1) || (h == -1)) && ((flags & 64) != 64)){
         flags = Toolkit.getDefaultToolkit().checkImage(img, -1, -1, null);
         w = Math.min(img.getWidth(null), maxW);
         h = Math.min(img.getHeight(null), maxH);
      }
      if ((flags & 64) == 64){img = null; ok = false; }
      else ok = true;
   }

   private void initp(Image i){
      if (i != null){
         img = i;
         w = Math.min(img.getWidth(null), maxW);
         h = Math.min(img.getHeight(null), maxH);
         checkFlags();
      }
      else ok = false;
   }

   public void paint(int x, int y, Graphics g){
      super.paint(x, y, g);
      if (pd == null) pd = new PicDraw(this, base);
      int w = Math.min(img.getWidth(pd), maxW);
      int h = Math.min(img.getHeight(pd), maxH);
      g.drawImage(img, x, y, w, h, pd);
      g.drawRect(x, y, w, h);
   }

   public void prep(Color c, Font f, Graphics g){
      checkFlags();
   }

   public void activate(View from){
      int w = img.getWidth(null);
      int h = img.getHeight(null);

      if ((w > maxW) || (h > maxH)){
         Frame win = new Frame("S.I.D. - expanded picture");
         PictureCanvas pc = new PictureCanvas(img, win);
         win.add(pc); win.addWindowListener(pc);
         win.pack(); win.show();
      }
   }
```

```
   public void showProperties(int x, int y, Frame from){ /* do nothing */ }

   public boolean init(Text t, int textPos, Frame parent){
      FileDialog fd = new FileDialog(parent,"Insert a Picture");
      fd.show();

      String dir = fd.getDirectory();
      String image = fd.getFile();
      if ((dir == null) || (image == null)) return false;

      Image i = Toolkit.getDefaultToolkit().getImage(dir+image);
      initp(i);
      parent.toFront();
      return ok;
   }

   public void load(DataInputStream f)throws IOException{
      super.load(f);
      int len = f.readInt();
      byte[] pixdata = new byte [len];
      f.readFully(pixdata,0,len);
      img = Toolkit.getDefaultToolkit().createImage(pixdata);
      ok = true;
   }

   public void store(DataOutputStream f)throws IOException{
      super.store(f);
      try{
         ByteArrayOutputStream out =
            new ByteArrayOutputStream(img.getWidth(null) *  img.getHeight(null));
         GIFEncoder encoder = new GIFEncoder(img);
         encoder.Write(out);
         f.writeInt(out.size());
         out.writeTo(f);
      }
      catch (AWTException e){
         throw new IOException("too many colors");
      }
   }
}
```

3.4. Hyperlinks

Hyperlinks consist of two parts. The target of a hyperlink is a *MarkElem* which contains a fingerprint identifying the element inside the text. This target may be referenced by an arbitrary number of *HyperLinkElems*. The view of the text containing the MarkElem must implement the *Hyperlinkable* interface in order to give the HyperLinkElem access to its target elements.

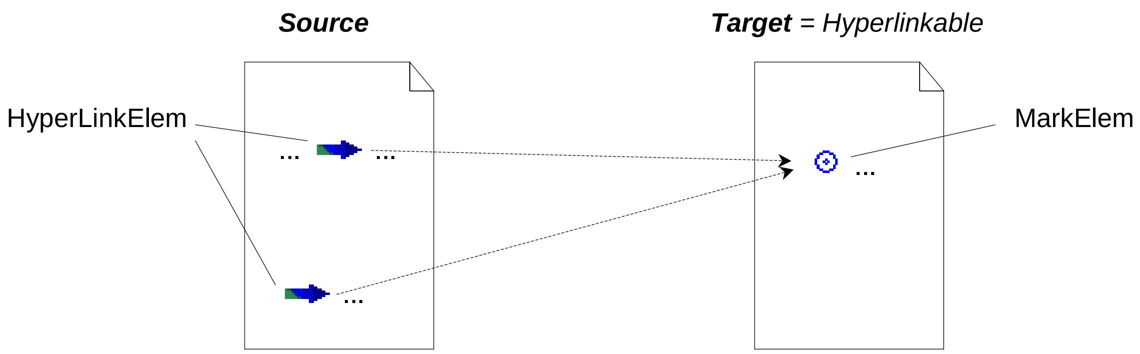


Figure 12: Structure of Hyperlinks

When following a HyperLinkElem, at first the hosting text is loaded (its name is stored in the HyperLinkElem) and asked for the element with the fingerprint (also stored in the HyperLinkElem). This is done via the Hyperlinkable interface. When the MarkElem is found, the hosting view is asked to show the element which may include actions like scrolling

(*ScrolledTextView*) or switching text to the foreground (*BoxView*).

3.4.1. MarkElem

MarkElems are passive elements with no user interaction and only one editable property – a description string which is displayed when a corresponding HyperLinkElem is selected. The unique identification of the hypertext target inside the text is guaranteed by a fingerprint which is initialized with the system time in the constructor and cannot be changed later.

attributes:

private long *fingerprint*	Fingerprint, stores the time of creation of the element, millisecond precision.
protected String *explanation*	Description string.

Hyperlinks are interactively inserted via a simple clipboard mechanism. A newly created MarkElem may be inserted into a global clipboard (represented by the static attributes copy and htext).

static attributes:

public static MarkElem *copy*	Clipboard for hypertext target exchange.
private static String *htext*	Name of the hosting text of the copied element.

methods:

void *paint*(int x, int y, Graphics g)	Draw the element (Circle plus Dot).
long *getFingerPrint*()	Get the fingerprint (read-only attribute).
void *setExplanation*(String e)	Set the description string.
String *getExplanation*()	Get the description string.
boolean *init*(Text t, int textPos, Frame parent)	Do nothing, init already done in constructor.
void *activate*(View from)	Do nothing, MarkElems are passive elements.
void *prep*(Color c, Font f, Graphics g)	Do nothing, MarkElems always look the same.
void *showProperties*(int x, int y, Frame parent)	Open properties dialog to edit the explanation string (class *MarkPropertiesDialog*).
String *toString*()	Get string representation of the element (used for debugging).
void *store*(DataOutputStream f)	Store element to the output stream *f*.
void *load*(DataInputStream f)	Load element from the input stream *f*.
static methods	
static void *copy*(MarkElem m, String text)	Copy element *m* to global clipboard, parameter *text* contains the name of the hosting text.
static HyperLinkElem *paste*()	Return new HyperLinkElem referencing the copied MarkElem.
static Vector *createMarkElemList*(Text t)	Create a vector of all hyperlink targets in the given text. Used for editing purposes – an overview of all targets may be useful for persons editing the source text and inserting links into other texts.

3.4.2. Hyperlinkable

The *Hyperlinkable* interface must be implemented by views containing MarkElems. A Hyperlinkable object must be able to locate a MarkElem with the given fingerprint in the displayed text and to show a given MarkElem on the screen (by scrolling, etc.). Additionally a Hyperlinkable object must support iteration over all hosted MarkElems.

```
public interface Hyperlinkable {

  public MarkElem findTarget(long fingerprint);

  public void show(MarkElem me);

  // MarkElem iterator
  public MarkElem firstTarget();
  public MarkElem nextTarget();
}
```

3.4.3. HyperLinkElem

A HyperLinkElem references a MarkElem in a Hyperlinkable (any text displayed by a view implementing the Hyperlinkable interface). The reference consists of the path to the file containing the text in which the referenced MarkElem is residing and the fingerprint of the MarkElem.

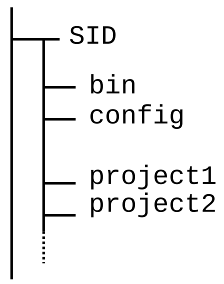
3.4.3.1. Relative Paths

SID documents may be browsed/edited locally or by an applet on the internet. When browsing remotely via the applet, a server process managing the SID directory structure makes the appropriate documents available. Any path in HyperLinkElems is stored relative to the SID root directory. Projects have to be stored in separate subdirectories of the SID root directory.

e.g.:

SID is installed at `C:/Prg/SID`

Documentation files are stored in subdirectories at `C:/Prg/SID`



A relative path like `"project1/classes/classADoc.htxt"` refers to the absolute path `C:/Prg/SID/project1/classes/classADoc.htxt` when browsing locally and to the URL `http://SIDSrv/project1/classes/classADoc.htxt` when browsing remotely and the SID server process is running on the server *'SIDSrv'* (this information is stored in the applet starting up the SID remote browser).

The class *NameSpace* is used to determine whether the browser is running with local or remote configuration. When SID is started, either the starter class *SIDStarter* (local mode) or the starting applet (remote mode) set the 'remote' flag in the global *NameSpace* variable and store additional information like the server name or the root directory in order to complete the relative path.

3.4.3.2. Attributes

HyperLinkElems may display an icon or an optional string.

protected String *targettext*	Relative path to the target text.
protected long *fingerprint*	Fingerprint to the referenced MarkElem.
protected String *text*	Optional string to be displayed.
protected boolean *showString*	Flag that indicates whether the string *text* should be displayed or not.
protected Image *ic*	Default icon to displayed, if no string is shown.
public static final int *W* = 22, *H* = 12	Dimensions of the default icon.

3.4.3.3. Methods

constructors	
HyperLinkElem()	Default constructor.
HyperLinkElem(String target, long fingerprint)	Create element with reference to text *target* and MarkElem with given fingerprint. Display the default icon.
HyperLinkElem(String target, long fingerprint, String text)	Create element with reference to text *target* and MarkElem with given fingerprint. Display given text.
boolean *init*(Text t, int textPos, Frame parent)	Do nothing, all necessary information is passed to the constructors.
accessors	
void *setTarget*(String name)	Set the target text path, the path is converted to a relative path.
String *getTarget*()	Get the target path. It is converted to an absolute path depending on the browser mode (local or remote).
void *setText*(String str)	Set the displayed descriptive text.
String *getText*()	Get the displayed descriptive text.
void *toggleStatus*()	Switch display status from 'text' to 'icon' and vice versa.
boolean *getStatus*()	Get the display status.
string operations	
static String *trimString*(String str)	Replace any `'/'` or `'\'` characters with the current separator.
static String *getPrefix*()	Retrieve prefix depending on browser mode: local: SID root directory remote: `'http://'` + SID server name
static String *getAbsolute*(String path)	Build an absolute path out of the given relative path.
static String *getRelative*(String path)	Build a relative path out of the given absolute path.
interaction	
void *activate*(View from)	Follow the hyperlink, show the text containing the target and display the referenced MarkElem.
void *showProperties*(int x, int y, Frame parent)	Open a dialog displaying the elements properties (class *HyperLinkPropertiesDlg*).
String *helpMsg*()	Display a descriptive help text.
Hyperlinkable *getTargetHost* (View parent)	Retrieve the referenced host text.

display	
void *paintString*(int x, int y, String s, Graphics g)	Display the string (only called when status is 'text').
void *paint*(int x, int y, Graphics g)	Draw the element (icon or string).
void *prep*(Color c, Font f, Graphics g)	Prepare to be displayed with given font and color (only relevant when status is 'text').
serialization	
void *store*(DataOutputStream f)	Store element to output stream *f*.
void *load*(DataInputStream f)	Load element from input stream *f*.

Discussion