Bachelor Thesis

TimeVis
Visualizing Temporal Data using prefuse

ausgeführt am Institut für
Softwaretechnik und Interaktive Systeme
an der Technischen Universität Wien

von
Peter Weishapl
0304333

Wien, im März 2007

Contents

1 Introduction

The aim of this thesis was to extend and generalize *PlanViewer*, a Software Prototype applying *PlanningLines* [2] to *MS-Project* and *Asbru* plans. As *PlanViewer* has been generalized to not only visualize plans, but any kind of temporal information, and to emphasise the difference between the old *PlanViewer* and the new, extended version, the new prototype is called *TimeViewer.* As a byproduct of *TimeViewer* an *API* called *TimeVis* has been developed. *TimeVis* enables developers to quickly develop applications that need to display any kind of temporal information. Both, *TimeVis* and *TimeViewer* have been developed using *Java Swing, Java 2D* and *prefuse.* Though one goal was to evaluate *prefuse*, as to find out how well it could be used to visualize temporal data, *TimeVis* has been designed to be as independent as possible from *prefuse*, so that it could be used with anything built on *Java 2D* and *Java Swing.* This however doesn't mean that *TimeVis* will work "out of the box" with anything except *prefuse*, but it should be pretty easy to adapt, as the dependence on prefuse has been kept to a minimum.

1.1 PlanningLines

PlanningLines is a visualization technique that allows for representing temporal uncertainties and aims at supporting project managers in their difficult planning and controlling tasks. A *PlanningLine* is essentially a glyph used to visualize an interval, for example an activity in a project schedule. *PlanningLines* therefore extend the concept of *GANTT charts*[1] by providing visualization not only for starting and finishing time of an activity, but for:

- start interval
  - earliest starting time [EST]
  - latest starting time [LST]
- end interval
  - earliest finishing time [EFT]
  - latest finishing time [LFT]
- duration
  - minimum duration [minDu]
  - maximum duration [maxDu]

You can see an example of a *PlanningLine* in Figure 1.

Though PlanningLines work well with *GANTT charts*, they can be used to visualize future activities in any context. A popular example are medical treatment plans. [3]

[1]See http://en.wikipedia.org/wiki/Gantt_chart (accessed on April 30th, 2007)

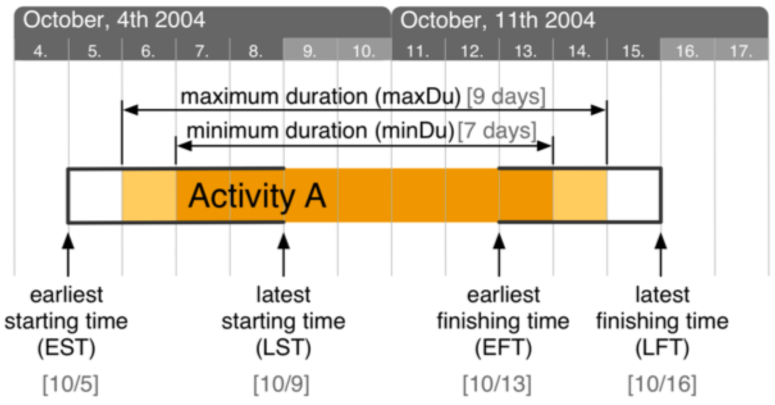
October, 4th 2004
October, 11th 2004
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
maximum duration (maxDu) [9 days]
minimum duration (minDu) [7 days]
Activity A
earliest starting time (EST) [10/5]
latest starting time (LST) [10/9]
earliest finishing time (EFT) [10/13]
latest finishing time (LFT) [10/16]

Figure 1: Example of a PlanningLine

1.2 From PlanViewer to TimeViewer

The goal of the *PlanViewer* application was to prove the concept of *Planning-Lines* in a real application. Another aim of the thesis was the evaluation of open-source toolkits intended for *Information Visualization* regarding their operability in the domain. Toolkits used in the *PlanViewer* prototype are the *InfoVis* and *prefuse* toolkits. [5] Figure 2 shows a screenshot of the *PlanViewer* prototype.

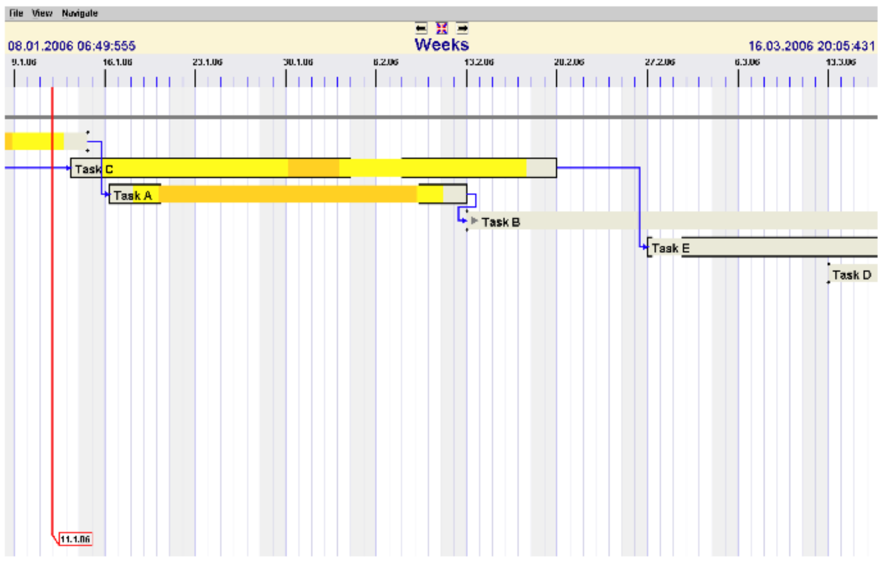

Figure 2: PlanViewer Prototype

As mentioned before, the goal of this thesis was to enhance the existing *PlanViewer*; particularly the following requirements should be met:

- Provide an *Overview Plus Detail*[2] interface, whereas the overview provides a *Range Slider* control[3] to navigate (pan and zoom) on the detail view. See figure 3, where the detail view is displayed in the top and the overview in the bottom part of the window. The overview also incorporates the *Range Slider* control, which is placed to focus the detail view on "Sub Task D". The *Overview Plus Detail* pattern has the advantages that the user can always see the whole plan in the overview, while looking at the details in the detail view. The *Range Slider* lets the user see at a glance which part of the whole plan is displayed in the detail view. Changing the detail view with the entire plan on view is also preferable to change the detail view without knowing where we are and what's outside the current view.

- Provide a *Fisheye View* for the temporal axis and enable the user to adjust the fisheye intensity and eventually to turn the fisheye on and off. A *Fisheye View* shows an area of interest near a focus point quite large, whereas the area far away from the focus point is shown in less detail. You can see an example of a *Fisheye View* in figure 3 in the detail view, where the focus point lies in the center and the time around the February 26 is shown in greatest detail.[4]

- Make it possible to visualize multiple plans beneath each other, so that it's easy for the user to compare them.

- The interaction and navigation, especially the *toolbar*, needs to be improved in order to enhance usability.

- It should be possible to search for activities.

- Extract a *(TimeVis) API* from the *TimeViewer* prototype to make it easy for developers to build applications with *Java Swing* and *prefuse* to visualize any kind of temporal information.

Instead of advancing the existing *PlanViewer* prototype, it has been rewritten from scratch. Though a couple of routines and lots of ideas have been copied or taken over from the existing prototype, rewriting was considered superior to advancing, due to the following reasons:

- *PlanViewer* has been written using an alpha version *prefuse*. Time has gone by and by the time of writing this thesis there is already a beta version of *prefuse* available. The alpha version not only has more bugs and a worse design, it also is outdated and not supported anymore, which means that the beta version is seminal and the alpha not. Besides nobody wants to learn an outdated technology, so the beta version is the way to go.

- *PlanViewer* uses the *InfoVis* library, which may be great, but in this context it's simply not necessary. It has been in *PlanViewer* since it's

[2]Also referred to as overview-detail in this thesis. See [6] for a description of this user interface design pattern.

[3]A Range Slider is a control that lets you input two values, an upper and a lower bound. In our case the bounds are dates lying within the bounds of the overview.

[4]For more information on *Fisheye Views* see http://www.infovis-wiki.net/index.php/Fisheye_View (accessed on April 30th, 2007)

early versions and has later been considered dispensable, but actually it has never been eliminated. Rewriting avoids the need to get rid of the *InfoVis* library.

- *PlanViewer* has a very interweaved class structure, which means that dependencies between classes and packages are very numerous and undirected and therefore hard to grasp. There are also a lot of static methods in these classes and even classes with static methods only. Furthermore there are a lot of relics, like *InfoVis*, which have never been removed or refactored. These all factors make it very hard to change, extend or even understand the program code. Rewriting avoids this burden.

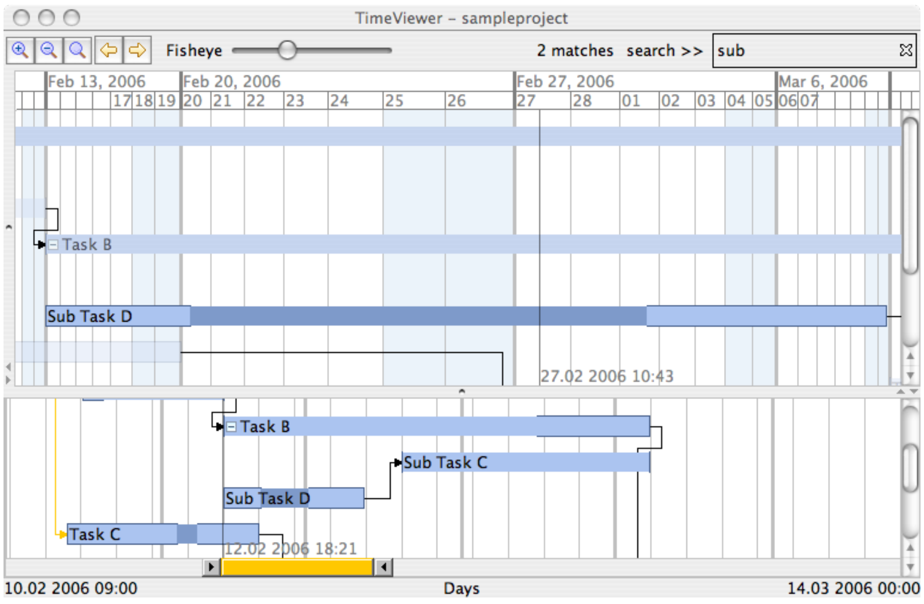

Figure 3: TimeViewer prototype

2 Information Visualization

> *Information Visualization* is the scientific discipline to display abstract data in a useful, clear, and comprehensible way.[5]

This sentence captures 3 important terms that need to be understood to understand how *Information Visualization* works: *data*, *information* and *visualization*. These terms will be described in this chapter and lay the foundation for understanding of how the *process* of *Information Visualization* works in visualization software, namely how to design software to display data in “a useful, clear and comprehensive way”. Moreover, how *prefuse* facilitates this process.

[5]see [5] on page 34

2.1 Data, Information and Visualization

In general, data is just a representation of something - numbers, letters, and dates, anything - suitable for communication, interpretation or processing. Which means data has no meaning; someone must interpret it. So interpreting is essentially assigning meaning to data. That's where information comes in. Information is data with an associated meaning. Assigning meaning to data is possible through interpreting data in a particular context. A combination of letters can be a description of a task; a combination of dates can be the interval in which the task should be executed.

Information Visualization is the process of displaying data to enable viewers to see, browse and understand the information. An important aspect of *Information Visualization* in computer programs is *interactivity*. Viewers do not only view, but also interact with the program. They not only need to be able to explore and search data and information, but also to accommodate the visualization to their current needs, for instance to zoom in to reach more details or zoom out to gain a broader view, highlight important aspects or expand and collapse composite data structures or even change the whole appearance of the visualization, like switching from 2D to 3D view or simply changing colors.

Developing software that fulfills these requirements is a complex task. A lot of research has been made in this area and different techniques for visualizing data and information has been developed. The technique used in the *TimeViewer* and *PlanViewer* projects is called the *Information Visualization Reference Model* also known as *InfoVis Pipeline*.

2.2 Displaying Data: The InfoVis Pipeline

The InfoVis Pipeline is a software architecture pattern that breaks up the process of Information Visualization into a series of discrete steps. Each step utilises the results of the previous step to eventually transform raw data into a concrete visualization. This process is illustrated in Figure 4.

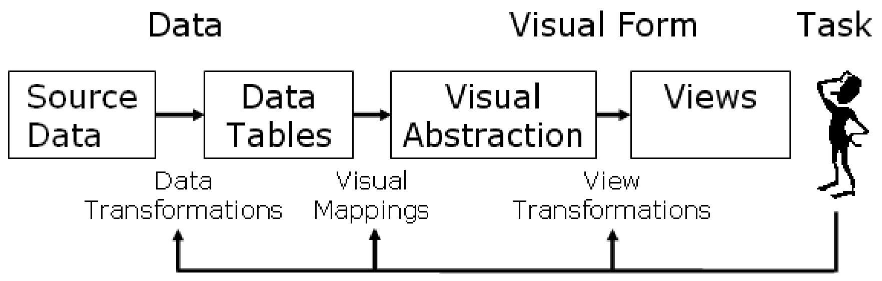

Figure 4: The *InfoVis Pipeline*. Source data is mapped into data tables that back a visualization. These backing data tables are then used to construct a visual abstraction of the data, modeling visual properties such as position, color, and geometry. The visual abstraction is then used to create interactive views of the data, with user interaction potentially affecting change at any level of the framework. [1]

The first step is to load the data and therefore transform any proprietary format into a generic, table like structure called *data tables*. The *source data* can be in any format, but after the transformation the data is essentially transformed into a collection of tables (or matrices) with rows, columns, and values potentially representing references to other tables to support more complex structures like *graphs* and *trees*.

Through *visual mappings, visual abstractions* are then created, a data model, which refers to the original *data tables* and includes visual features like color, size, position, or shape. The *visual abstractions* include all information necessary to render the data.

The actual rendering is done after *view transformations*. This step is necessary for instance to support panning or zooming, which in turn modifies the size and position of the *visual abstractions* for correct rendering according to the current view.

User interaction is supported at any step in this process. For example, the user could pan a view through a *Scroll Bar*, therefore modifying the *view*. The user interface could also provide a search facility that could highlight items by changing its color, therefore modifying the *visual abstractions* of the found data items.

2.3 prefuse[6]

prefuse is an implementation of the *InfoVis Reference Model* or *-Pipeline* mentioned above. It's built on *Java Swing* and *Java2D*. Figure 5 shows the relations between *prefuse* and the *InfoVis Pipeline*.

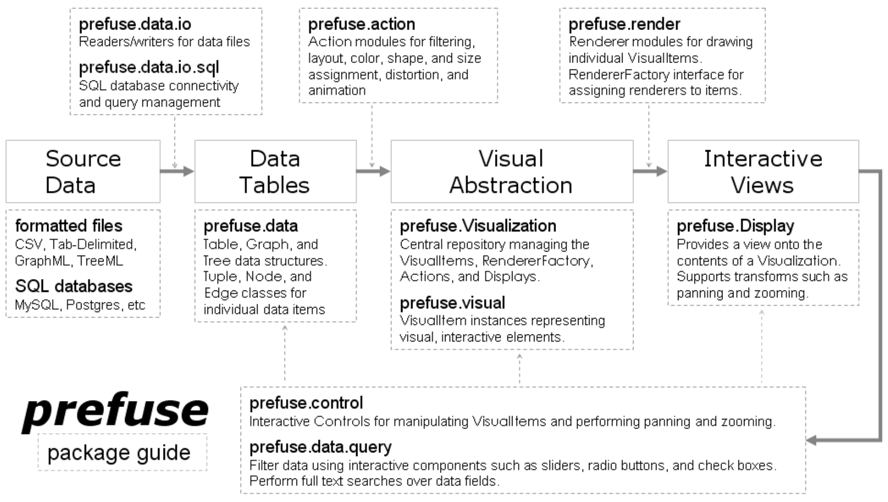
prefuse.data.io
Readers/writers for data files
prefuse.data.io.sql
SQL database connectivity and query management
prefuse.action
Action modules for filtering, layout, color, shape, and size assignment, distortion, and animation
prefuse.render
Renderer modules for drawing individual VisualItems. RendererFactory interface for assigning renderers to items.
Source Data
Data Tables
Visual Abstraction
Interactive Views
formatted files
CSV, Tab-Delimited, GraphML, TreeML
SQL databases
MySQL, Postgres, etc
prefuse.data
Table, Graph, and Tree data structures. Tuple, Node, and Edge classes for individual data items
prefuse.Visualization
Central repository managing the VisualItems, RendererFactory, Actions, and Displays.
prefuse.visual
VisualItem instances representing visual, interactive elements.
prefuse.Display
Provides a view onto the contents of a Visualization. Supports transforms such as panning and zooming.
prefuse
package guide
prefuse.control
Interactive Controls for manipulating VisualItems and performing panning and zooming.
prefuse.data.query
Filter data using interactive components such as sliders, radio buttons, and check boxes. Perform full text searches over data fields.

Figure 5: The *prefuse* package guide depicting prefuse packages and classes and their relations to the InfoVis Pipeline [1]

The *prefuse.data* package provides *Table*, *Graph*, and *Tree* classes for representing data. Table rows are represented by the *Tuple* interface, while the *Node* and *Edge* interfaces represent the members of graph and tree structures. The *Graph* and *Tree* classes are implemented using *Table* instances to store the node

[6]see prefuse.org

and edge data. These data structures match to *data tables* mentioned in chapter 2.2. The *prefuse.data.io* package provides classes for reading and writing table, graph, and tree data from formatted files.

The *Visualization* class acts as the *visual abstraction* as described above. For any *Tuple*, *Node*, or *Edge* added to the *Visualization*, a corresponding *VisualItem, NodeItem,* or *EdgeItem* instance is created. These classes, found in the *prefuse.visual* package, extend the *Tuple* interface and provide access to both the visual attributes and the underlying data.

The *visual mappings* (see chapter 2.2) are performed through *Action* classes located in the *prefuse.action* package. These classes manipulate properties of *VisualItems* like visibility, position, size and color. Subclasses of *Actions* are typically layouts, animators or highlighters. A number of prebuilt implementations can be found in the *prefuse.action* package and it's sub-packages. *Actions* can be combined sequentially as in a *ActionList* to support flexibility of visualizations and reusability of *Action* implementations.

The *prefuse.render* package contains *Renderers,* which read the properties of *VisualItems* and do the actual painting based upon this data. Different *Renderers* and *VisualItems* can be combined in any way, which again supports flexibility and reusability. *prefuse* provides *Renderers* for drawing various shapes, labels, and images out of the box.

The *Display* class acts as a camera onto the contents of a *Visualization.* The Display is responsible for drawing visible items (by using appropriate *Renderers*) and can be panned, zoomed, and rotated as desired. Multiple *Display* instances can be associated with a single *Visualization*, enabling multi-view configurations, including *Overview Plus Detail* views.

The *prefuse.controls* package provides *Controls* that can be registered with *Display* instances and be used to process mouse and keyboard actions performed by the user. *prefuse* provides pre-built *Controls* for selecting items, dragging items around, and panning, zooming, and rotating the *Display.* It's easy to write custom *Control* implementations by extending the *ControlAdapter* class.

Thanks to *prefuse's* unified data structure, essentially based on the *Tuple* and *TupleSet* interfaces, querying data and providing various interactive search and filter facilities to the user is a snap. Most of the classes supporting data queries can be found in the *prefuse.data.query* and *prefuse.data.search* packages.

3 TimeVis API

TimeVis provides a framework that enables *Java* developers to build applications to visualize any kind of temporal data without much effort. The central element is the *BasicTimeScale* class, which handles the mapping between pixels on the screen and dates represented by this pixel. The *AdvancedTimeScale* and *FisheyeTimeScale* classes respectively extend the functionality of the other classes. All timescale implementations use the *TimeUnitProvider* for acquiring their best-suited *TimeUnit.*

3.1 Timescales

The main responsibility of timescales is to provide a mapping between pixels and dates. That is, to find a date for a given pixel and vice versa.

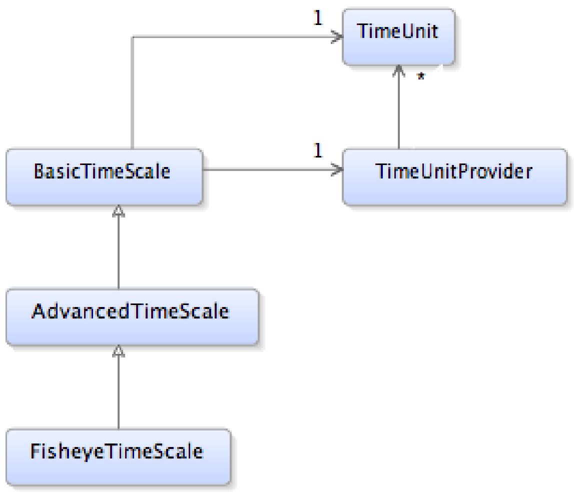

Figure 6: TimeVis core classes

BasicTimeScale provides the following methods to accomplish these tasks: *getDateAtPixel* and *getPixelForDate*. The implementation is quite straightforward. *BasicTimeScale* maintains a start date and a number of milliseconds that should be used per pixel. For example if the start date is exactly *8 March 2007, 00:00:00*[7], *getDateAtPixel(0)* would return exactly this date, and *getPixelForDate* with the start date as the input, would return 0, of course. What if you want a date for a pixel other than the first one, like, say number *60*? The returned value of course depends on the number of milliseconds to use per pixel. Suppose we have told *BasicTimeScale* to use 1000 milliseconds, which is one second, per pixel. The returned date would be *8 March 2007, 00:01:00* - one minute after the start date. The implementation, again, is easy: Take the start date and add the number of milliseconds per pixel multiplied with the given pixel, in this case 60.

At a glance, it seems that the mapping between pixel and date is 1 to 1, one pixel for exactly one date. But if you look carefully, you'll notice that in truth every pixel represents an interval, a range of dates: the number of milliseconds per pixel - not a single milliseconds. This is not a problem, as long as you don't need to correctly tell the user, which date is represented by a single pixel. The user often needs a correct time, not a possible range of milliseconds. Let's look at the *TimeViewer* prototype (see page 13) as an example. Here, activities that have an exactly defined start and end date, are visualized. Suppose a start date of an activity is some day at *01:00:00* o clock and the start date of a *BasicTimeScale* is on the same day at *00:00:01*, nearly one hour earlier, but not exactly. Now, suppose the number of milliseconds per pixel is 60000, one minute. The start of our activity would be rendered at pixel 59 - as returned by *getPixelForDate* - because this pixel represents the interval *00:59:01* to *01:00:01*.

[7]Milliseconds in dates are always omitted in this thesis, except when it is useful for explaining a concept. Generally you can assumed that all units omitted are 0.

This is all right again, as long as you're not going to ask the *BasicTimeScale* for the date at pixel 59 and present it to the user. The user would certainly be confused because the *Activity* all of a sudden would end at *00:59:01*, which would seem very weird.

To solve the problem mentioned above we need to *adjust* dates to avoid confusing the user. But how and against what should the date be adjusted? It certainly depends on the resolution[8] of the timescale. If one pixel represents exactly one minute, do we simply want to adjust dates to minutes, so that *00:59:01* becomes *01:00:00*? Well, sometimes, but not necessarily always. We would always have to ask the user how to adjust, and that would be cumbersome. Another problem is, if one pixel for instance represents 128931273 milliseconds, to what unit should be adjusted? Seconds, days, months? What are the units? In some cases it could make sense to have units of 5 seconds or quartals.

3.2 Time Units

The idea behind time units is to (of course) not ask the user how to adjust dates, but to tell the user how the dates are currently adjusted, or, more comprehensible for the user, how accurate the current display is. The other aspect is to let the developer define which time units are available[9] and let the timescale decide which time unit to use, depending on the current resolution, or zoom level. In technical terms, the *BasicTimeScale* uses the *TimeUnitProvider* to get a *TimeUnit* suiting the current resolution to tell the user about the accuracy of the current display.

The easiest way to understand a *TimeUnit* is by looking at how it's visualized in the context of a timescale, as you can see in figure 7. A *TimeUnit* is rendered using vertical lines. In the figure you can see two different *TimeUnits*. First the timescale's current *TimeUnit*, which is represented by the thin lines, and the next longer *TimeUnit* provided by the *TimeUnitProvider*, which is represented by the thicker lines. You can see in the figure that the shorter *TimeUnit* represents days, and the longer weeks.

By now we have defined what the *TimeUnit* is and what it's used for. But what are the responsibilities of the *TimeUnit* class? First, it can provide dates that *fit* into it. For instance *8 March 2007, 00:00:00* fits into the time unit days as well as into seconds or hours, but not into months. The *8 March 2007, 01:00:00* doesn't fit into days or any unit greater than that but fits into anything smaller than days. The *1 January 2007, 00:00:00* fits into virtually every time unit, except decades. By providing these dates, the time unit can be painted like the vertical lines in figure 7. Also, the timescale can use this information to adjust the pixel-to-date mapping to the current time unit. The *getDateAtPixel* method of the *BasicTimeScale* for example adjusts dates before returning them, so that for pixels representing a date that fits into the current *TimeUnit* this date that is returned and not any other date of the represented interval. This adjustment ensures that the date *10.03.2007 00:00* displayed in figure 7 represents exactly the date indicated by the vertical line, and not *09.03.2007 23:55*, which could be represented by the same pixel and therefore

[8]The resolution is determined by the number of milliseconds per pixel. The smaller this number, the higher the resolution.

[9]There is a default set of TimeUnits available, so the developer must not necessarily worry about defining own *TimeUnits*.

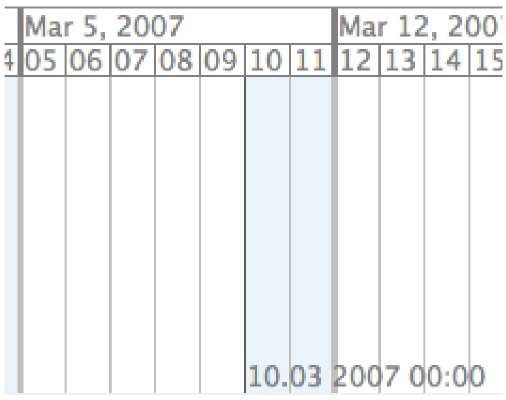

Figure 7: Visualized timescale with a time unit of one day. This time unit is rendered using the thin, gray vertical lines.

would not be wrong from a technical point of view but would certainly confuse the user.

Another responsibility of the *TimeUnit* is, to provide *DateFormats* for formatting dates according to the accuracy of the unit. You can see in figure 7a use of all 3 DateFormats provided:

- The **short format** is used by the *TimeScaleHeader* class to format dates in the header area when space is short.
- The *TimeScaleHeader* also uses the **long format** to format dates when there is more space.
- The *MouseHandler* class paints a vertical line at the current mouse position and displays the date at this position, formatted with the **full format**.

As available time units are typically defined by the developer it is important to note that a *TimeUnit* object must be based upon a unit provided by a field of the *Calendar* class. So the *TimeUnit* cannot be of any length, but can only be a multiple of the *Calendar* field it is based upon, like 5 seconds, 3 months, or one hour. This is because finding a date fitting within a *TimeUnit* is not a straightforward task and utilises functionalities of the *Calendar* class for which the *Calendar* field of the *TimeUnit* is required. The fact that months and years differ in length further complicates this process.

3.3 Using TimeVis

TimeVis has been designed to work in conjunction with *Java Swing* and *prefuse*. However, the core classes (see figure 6) are completely independent of these libraries and you can build your application using these classes, with any GUI and visualization toolkit you like. However, there are a lot of classes in the framework that are built to be used with *prefuse* or *Java Swing,* which makes it preferable to use these toolkits*:*

- The *RangeAdapter* class simplifies the process of building overview-detail views by implementing a *Swing BoundedRangeModel* that can be used by the *prefuse JRangeSlider* class.

- The *at.ac.tuwien.cs.timevis.actions* package contains *Swing Action* classes used to interact with various timescale implementations, the *RangeAdapter* class, and prefuse *Displays.*
- The *at.ac.tuwien.cs.timevis.ui* package on the other hand provides *Swing* components and classes like the *StatusBar* for displaying the start- and end date and the current *TimeUnit* of a given *AdvancedTimeScale,* or the *TimeScaleHeader* painting a header for a timescale.
- The *at.ac.tuwien.cs.timevis.prefuse* package and its sub-packages include classes for laying out, rendering and interacting with temporal data.

There are also some demos and of course the *TimeViewer* prototype demonstrating the use of some of these classes in the *at.ac.tuwien.cs.timwvis.demos* package.

Although *TimeVis* is only used with *prefuse* at the time, it has been designed to be independent of any library. That is why zooming, panning and distortion using *prefuse*'s *Display* class is not supported by *TimeVis* and may lead to unexpected results. For example the *TimeLayout* lays out temporal data along the x-axis utilising the *BasicTimeScale*'s pixel-date mapping. This requires that the *Display*'s view coordinate system does not differ from the world coordinate system for the x-axis, as this would corrupt the layout. Since *TimeLayout* does not use the y-axis, it can be modified arbitrarily.

When using *TimeVis*, you have to decide which of the 3 timescale implementation to use, or even if it is feasible to write an own implementation based on one of the provided timescales. The most basic timescale - *BasicTimeScale* - is best suited for applications, where you have a single view on the data and don't need to know the interval of the timescale. If you want overview-detail views with a *Range Slider*, you need to know the end date of the timescale and therefore you have to use an *AdvancedTimeScale.* The *AdvancedTimeScale* also features automatic calculation of it's resolution, based on it's interval an the width used to display this interval. It's also convenient if you want to adjust the view when the user resizes the window. If you also need a fisheye distortion you have no choice but use the *FisheyeTimeScale,* which inherits all features from the *AdvancedTimeScale* and has facilities to adjust the distortion intensity.

4 TimeViewer Prototype

The *TimeViewer* prototype is a remake of the *PlanViewer* prototype, incorporating the *TimeVis API* and enhanced according to the requirements mentioned in chapter 1.2. A few minor features of *PlanViewer*, like loading *Asbru* plans, have been omitted in the current version of the prototype.

4.1 User Interface

The *TimeViewer* UI consists of four main parts (see figure 3):

- The ***overview*** in the bottom area, which displays the whole interval occupied by the loaded activities. It also incorporates a *Range Slider* control used to modify the interval displayed by the detail view. It provides the big picture.

- The ***detail view*** above the *overview*, which displays the part of the *overview*, which is determined by the length and position of the *Range Slider* component used in the *overview.*
- The ***status bar*** at the bottom, which displays the start and end date of the *detail view* and it's current accuracy.
- The ***toolbar*** at the top that provides components to interact with the *detail view.* First, there are buttons to zoom and pan[10] and a *Slider* component to adjust the intensity of the fisheye distortion. Furthermore, there is a *search field* for searching activities, which will be highlighted when found.

Additionally, there is the ***menu bar***, which is used to open and insert MS-Project files and includes items that can also be found in the *toolbar.* Left of the detail view, there is a ***Tree*** component, displaying the hierarchical composition of the loaded schedule.

Most of the prototype's functionality is evident, either based on *TimeVis (*see chapter 3), *GANTT charts* or *PlanningLines*, there are however a few aspects worth mentioning:

- As described earlier, the *detail view* can be adjusted by changing the *Range Slider* provided by the *overview.* However, rarely schedules must be displayed which include activities that greatly differ in length. There could for example be an activity that lasts one month and consists of sub-activities lasting only a couple of minutes. To display this short activity accurately in the *detail view* you would have to reduce the length of the *Range Slider* to a fraction of the available length. The problem is, that the number of pixels of the available length limits this fraction. So, if you have a window width of 800 pixels, you can "only" make the *detail view* 800 times more accurate than the *overview* because you can only change the length of a *Range Slider* on a pixel-by-pixel base. For these rare cases, the possibility to pan and zoom the *overview* - which of course affects the *detail view* - has been included. Zooming can be achieved by pressing the right mouse button on the background and dragging the mouse up or down. Panning is accomplished by clicking on the background with the left mouse button and then dragging the mouse. Note, that this type of interaction is also possible with the *detail view*, but in turn only modifies the *Range Slider* and therefore is limited to the interval displayed in the *overview.*
- When typing into the search field, *TimeViewer* will highlight activities whose name contains the typed string by dimming all other activities that don't match the given string. Furthermore, activities containing found sub-activities are only a little dimmed to indicate that there are matching activities beneath it.
- *TimeViewer* maintains the interval of the *overview* when the window is resized, so that it never changes. As the *detail view* is dependent on

[10]Note that this is implemented by modifying the position and the size of the *RangeSlider* incorporated by the *overview.*

the length of the *Range Slider* and it's not reasonable to always adjust this length, the *detail view* naturally changes when the window is resized, which may seem odd to the user. However, the advantages are that the *overview* always shows exactly the whole interval and the *detail view* always stays in sync with the *Range Slider.*

4.2 Implementation

TimeViewer was implemented with *prefuse* (see chapter 2.3) and *TimeVis* *(*described in chapter 3) according to the principles of the *InfoVis Pipeline* mentioned in chapter 2.2. The *TimeViewer* core architecture with it's relations to the *InfoVis Pipeline*, *prefuse* and *TimeVis* are depicted in figure 8. The classes in the figure are laid out from top to bottom, where classes at the top relate to early steps in the *InfoVis Pipeline*, which naturally have a lot to do with data loading and representation. The further down the classes are, the more they are related to the user interface and therefore less to the data. The diagram has been divided into parts, each relating to a part of the *InfoVis Pipeline.* Note that the package structure is similar to the depicted parts. Classes belonging to *prefuse* have a gray background color, *TimeVis* classes are magenta, and *TimeViewer* classes blue.

The *MSProjectReader* class is used to read *MS-Project* files and transform it into an *ActivityGraph.* The *ActivityGraph* is a directed *prefuse Graph*[11] class and exists of nodes that are of type *Activity* and edges of type *ActivityRelation.* Instances of these two classes are managed in *data tables* the *ActivityGraph* and therefore must extend the prefuse classes *TableNode* and *TableEdge.*

ActivityNode and *ActivityEdge* are the visual abstractions of *Activity* and *ActivityRelation*, respectively. As defined by the *InfoVis Pipeline* they contain visual information in addition to the raw data. They too extend *prefuse* classes that are managed in *data tables* as well but also include support for visual information.

The most central class in any *prefuse* architecture is the *Visualization* class; in this case it's an *ActivityVisualization* instance. A *Visualization* is responsible for managing the mappings between source data and what is visualized on the screen. It maintains *Renderers* and a list of *Display* instances responsible for rendering of and interaction with the contents of the *Visualization.* It also manages a set of *Action* instances used for performing visual data processing such as position, size, and color assignment.

The *ActivityVisualization* is configured with an *IntervalLayout* used for laying out *ActivityNodes* along the x-axis and an *ActivityLayout* arranging *ActivityNodes* along the y-axis. It unsurprisingly uses an *ActivityNodeRenderer* for rendering *ActivityNodes* and an *ActivityEdgeRenderer* for rendering *ActivityEdges.*

The *ActivityDisplay* is a *prefuse Display,* which renders the contents of a given *ActivityVisualization* and handles interaction with it.

[11]This class implements a mathematical graph data structure as defined in the graph theory. See http://en.wikipedia.org/wiki/Graph_%28mathematics%29 (accessed on April 30th, 2007) for a description.

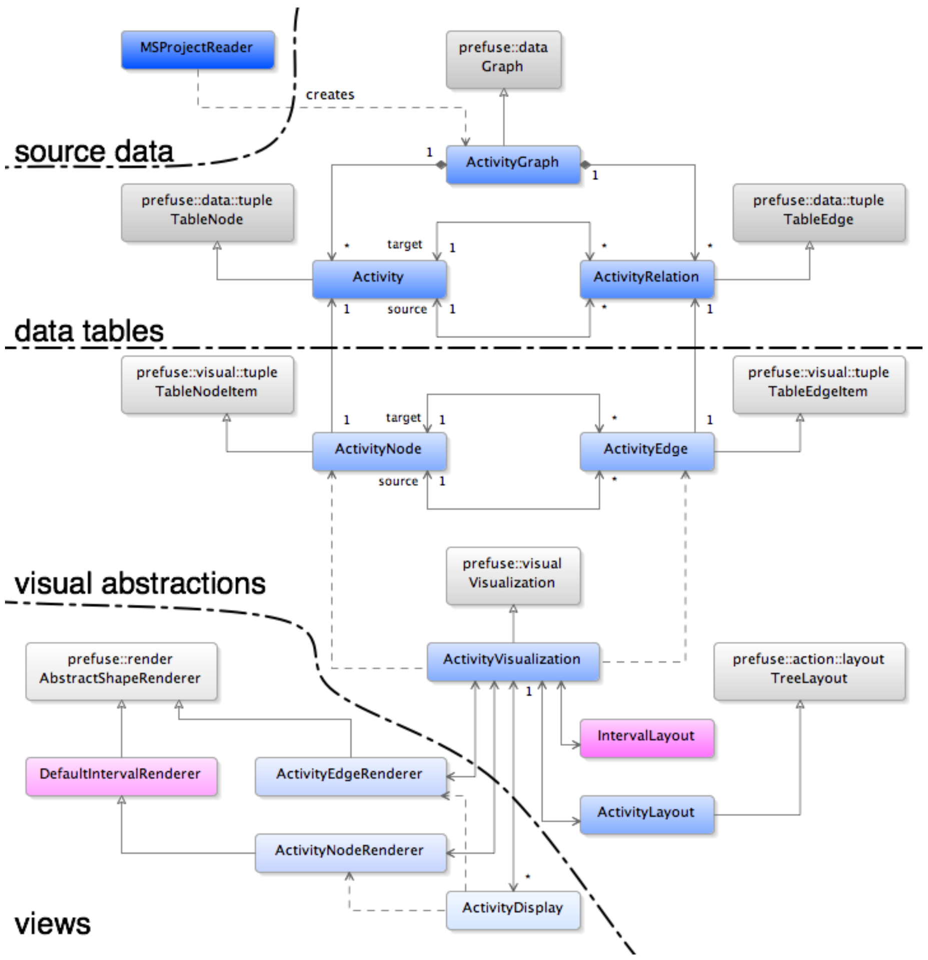

Figure 8: The *TimeViewer* architecture, how it maps to the *InfoVis Pipeline* and which *prefuse* and *TimeVis* classes are utilised.

5 Conclusions

In the course of this thesis I have developed *TimeVis*, an *API* for developing applications that visualize temporal data. *TimeVis* has originated from the development of the *TimeViewer* prototype, which is used to display time schedules consisting of activities and sub-activities. Both outcomes of the thesis are used in conjunction with *Java Swing, Java 2D* and *prefuse.*

5.1 On prefuse

prefuse is stated to be a toolkit for interactive information visualization and one goal of the thesis was to validate this toolkit. Generally, *prefuse* is a well structured and documented and it was quite easy to build the *TimeViewer* prototype using this library. However, it is still in beta phase and nothing has changed since one year. Additionally the prefuse manual still is a stub and therefore a developer needs to dig deep into the source code to fully understand it, which means the learning curve is quite steep. The great strength of *prefuse* is its flexibility. You can combine actions, layouts, animators, renderers, displays etc. in any way and it's possible to get great results very quickly. But in the case of *TimeViewer*, where it's very important to exactly layout activities - every pixel counts - the strengths of *prefuse* suddenly vanish. It seems that pixel-perfect layout just isn't supported and *prefuse* really doesn't seem to be built for such uses. Also the advantage of flexibility cannot be exploited by applications like *TimeViewer*, as one doesn't need to experiment a lot with different forms of visualizations and interactions and one form of visualization, once established, stays very stable over time, which means that the flexibility provided by *prefuse* really isn't needed. Therefore I think the effort to learn *prefuse* easily compensated the advantages it brought and I still think that by just using *Java Swing* and *Java2D* the effort would have been about the same and the result just as well. One annoying thing was that *prefuse* classes are quite hard to extend and customize, especially the *Visualization* class and classes concerning *data tables* like *Tuple* implementations. For example derived properties are not supported by *TableTuples* and therefore must be saved and kept up to date instead of just be calculated from other fields. An example is the real start in the *Activity* class, which is derived from the earliest- and latest start. I'm a fan of simple, plain old java objects[12] and promote its use wherever possible. *prefuse* on the other side is uses *data tables,* which means that data is not stored in objects, but in tables like in a relational database. The drawbacks of this are that *prefuse* can only work with it's own custom data structure and no behaviour can be implemented in the data objects. I think the use of these custom data structures is very cumbersome and it would have been better and easier if prefuse would be based on plain java objects. It is advocated that this data structure is so great because querying this data is so easy, efficient and generic. I believe that it makes no sense to build a proprietary data structure to mimic a relational database and I also think that querying plain objects can be easy, efficient and generic as well, for instance by using *Java Reflection*, and could save a lot of compatibility problems and learning effort. To summarize, *prefuse* is great for simple and fancy visualizations, but to be really useful for more complex, more specialized applications like *TimeViewer* it needs a better documentation and a better data structure, which is easier to learn, extend and customize.

5.2 On TimeVis

As *prefuse* is not considered an optimal toolkit for visualizing temporal data, *TimeVis* has been designed to be independent of it. It's pretty easy to build applications using *TimeVis*, but at the time it's quite hard to customize. Perhaps it should be easier to extend and customize timescale implementations. To achieve this, the proven principle of favoring composition over inheritance[13] should be applied to the *TimeVis* core classes to better support extensions and customizations. A possible structure for future versions of *TimeVis* is proposed in figure 9.

In future versions of *TimeVis* it should be possible to configure time units and it's date formats in a configuration file. Therefore the *TimeUnitProvider*

[12]http://en.wikipedia.org/wiki/Plain_Old_Java_Object (accessed on April 30th, 2007)
[13]see Item 14 in [4].

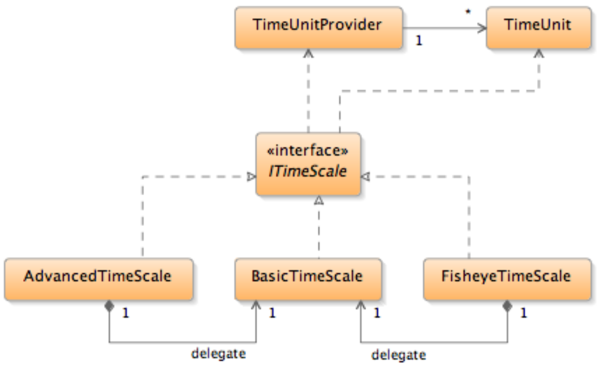

Figure 9: Proposed TimeVis core architecture for possible future versions

class is intended to be sub-classed, and it should be quite easy to write a *XML-TimeUnitProvider* which can be configured via a XML file and can be associated with a timescale.

5.3 On TimeViewer

TimeViewer is a great example of how to use *TimeVis* and *prefuse* to build a powerful application. It was built to verify *prefuse* and demonstrate the use of *PlanningLines*, which it does very well too. However, it is only a prototype and has its glitches and there are naturally a lot of things that could be improved:

- As mentioned in chapter 4, zooming and panning the detail view is performed through manipulating the *Range Slider* of the *overview*. When the *detail view* is much smaller than the *overview*, zooming and panning is very rough. This could be improved by changing the interval of the *detail view* instead of changing the *Range Slider* when panning and zooming. Then the *Range Slider* can keep track of the current interval and manage to stay in sync with it. Panning and zooming would then always be smooth, independent of the current intervals.
- When searching activities, the *detail view* could automatically focus on the area containing all found activities, if some found activities are outside the currently visible area. It is also possible to automatically expand parent activities of found sub-activities.
- The *overview* could better highlight the interval that is currently being displayed in the *detail view*.
- The item state of activities displayed in the *overview* and the detail could be synchronized. For example, when expanding a parent activity in the *detail view* it could be expanded in the *overview* as well.
- The prototype has a fairly bad performance as this is a prototype and performance has never been considered important during development.

Optimization should be fairly easy though, but it may be hard to profile the application and to find bottlenecks.

- Occasional hangs occur when zooming, resizing the window, or changing the fisheye intensity. The error seems to be somewhere in the painting code. Again this seems to be easy to fix but hard to find, especially because the hang-up is hard to reproduce.

References

[1] prefuse user's manual, May 2006.

[2] Wolfgang Aigner, Silvia Miksch, Bettina Thurnher, and Stefan Biffl. Planninglines: Novel glyphs for representing temporal uncertainties and their evaluation. Technical report, Institute of Software Technology & Interactive Systems, Vienna University of Technology, 2005.

[3] Wolgang Aigner. Interactive visualization of time-oriented treatment plans and patient data. Master's thesis, Vienna University of Technology, 2003.

[4] Joshua Bloch. *Effective Java Programming Language Guide*. Prentice Hall PTR, 2001.

[5] Andreas Fellner. Treating temporal uncertainties of complex hierarchical data visually. Master's thesis, Vienna University of Technology, 2006.

[6] Jenifer Tidwell. *Designing Interfaces*. O'Reilly Media, 2005.