

Treating Temporal Uncertainties Of Complex Hierarchical Data Visually

eingereicht von
ANDREAS FELLNER

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Magister rerum socialium oeconomicarumque
Magister der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec.)

FAKULTÄT FÜR INFORMATIK, UNIVERSITÄT WIEN
FAKULTÄT FÜR INFORMATIK, TECHNISCHE UNIVERSITÄT WIEN

Studienrichtung Wirtschaftsinformatik

Begutachter
ao. Univ.-Prof. Mag. Dr. Silvia Miksch

Wien, im Juli 2006

*Dedicated to my grandparents Anna and Alois.
Thank you for all the things you have
taught me, your faith,
and your love.*

Abstract

Generally, planning means to make predictions of the future. The more information about a certain endeavor and its environment flow into a plan, the more realistic it gets. But, as predictions are only assumptions, there is no guarantee that these predictions will actually come true. Therefore, plans should consider at least anticipated uncertainties, others are not planable anyway. The concept *PlanningLines* is specialized on treating temporal uncertainties in plans visually. This work is an attempt to evaluate *PlanningLines* in a modern visualization. Further, also the technical realization of such an *Information Visualization* is of special interest.

This thesis especially treats two different domains of planning: The management discipline *Project Management* which supports planning in organizations and the scientific discipline *Protocol Based Care* which provides medical treatment plans as protocols. Both, *Project Management* and *Protocol Based Care*, provide different concepts of handling plans. A common technique is the splitting of a whole plan into several manageable tasks that must be fulfilled in a specified sequence to meet the goal of an endeavor. That means, a plan is hierarchically decomposed, furthermore, it depends on time.

Especially, the factor time is afflicted with a lot of uncertainties which should be considered in a plan. A time schedule that does not work may endanger the progress of a plan or can even cause a complete fail. To minimize this risk, some techniques to consider temporal uncertainties within a plan are available. The concept *PlanningLines* treats such uncertainties by applying a set of time attributes to each task which are represented visually.

To evaluate *PlanningLines*, a prototype was developed which applies this concept to *MS-Project* and *Asbru* plans. Besides implementing the graphical notation of *PlanningLines*, the prototype also provides some general techniques of *Information Visualization*. The result is a modern and highly interactive application which demonstrates the power of *PlanningLines* in practice and illustrates a way of treating temporal uncertainties visually.

Kurzfassung

Planen bedeutet Vorhersagen für die Zukunft zu machen. Je mehr Informationen über eine Unternehmung und dessen Umgebung in einen Plan fließen, desto realistischer wird dieser. Da solche Vorhersagen aber nur Annahmen sind, gibt es keine Garantie für deren Eintreffen. Daher sollte ein Plan zumindest absehbare Unsicherheiten berücksichtigen, andere sind ohnehin nicht planbar. Das Konzept *PlanningLines* berücksichtigt zeitliche Unsicherheiten visuell. Diese Arbeit wendet *PlanningLines* in einer modernen Visualisierung an. Auch die technische Realisierung einer solchen *Informations-Visualisierung* ist von speziellem Interesse.

Diese Diplomarbeit befasst sich mit zwei Bereichen der Planung: Der Management Disziplin *Projekt Management*, welche sich mit der Planung in Organisationen beschäftigt und der wissenschaftlichen Disziplin *Protokoll-Basierte Therapiebehandlung*, die Behandlungspläne als Protokolle bereitstellt. Sowohl *Projekt Management* als auch *Protokoll-Basierte Therapiebehandlung* stellen verschiedene Konzepte der Planung zur Verfügung. Eine übliche Technik ist das Unterteilen eines Planes in kleinere Aufgaben die in einer definierten Sequenz erfüllt werden müssen um das Planungsziel zu erreichen. Das bedeutet, dass ein Plan hierarchisch unterteilt ist und einen zeitlichen Bezug hat.

Gerade durch den Faktor Zeit entstehen viele Unsicherheiten die in einem Plan berücksichtigt werden sollten. Ein nicht funktionierender Zeitplan kann den Ablauf eines Planes gefährden oder diesen überhaupt zum Scheitern bringen. Es gibt verschiedene Techniken um zeitliche Unsicherheiten in einem Plan zu berücksichtigen und damit dieses Risiko zu minimieren. Das Konzept *PlanningLines* behandelt solche Unsicherheiten indem jede Aufgabe mit einer definierten Menge von Zeitattributen versehen wird welche grafisch dargestellt werden.

Um das Konzept *PlanningLines* zu evaluieren wurde ein Prototyp entwickelt, der es auf *MS-Projekt-* und *Asbru-* Pläne anwendet. Neben der grafischen Notation wurden auch einige andere Techniken der *Informations-Visualisierung* implementiert. Das Ergebnis ist eine moderne und interaktive Applikation, welche die Stärken von *PlanningLines* in der Praxis veranschaulicht und einen Weg aufzeigt, wie mit zeitlichen Unsicherheiten visuell umgegangen werden kann.

Contents

Abstract	ii
Kurzfassung	iii
Acknowledgements	vii
1 Introduction	1
1.1 Motivation	1
1.2 Overview of the Thesis	2
2 State of the Art	4
2.1 Project Management	4
2.1.1 Overview	4
2.1.2 Definitions	5
2.1.3 Responsibilities of Project Management	6
2.1.4 Problems in Project Planning	9
2.1.5 Graphical Plan Representation - Charting Techniques	12
2.1.6 Software Tools	18
2.2 ProtocolBased Care	22
2.2.1 Overview	22
2.2.2 Clinical Guidelines and Protocols	23
2.2.3 The Guideline Representation Language Asbru	24
2.2.4 Graphical Representation of Treatment Plans	27
2.2.5 Related Projects	32
2.3 Information Visualization	34
2.3.1 Overview	34
2.3.2 Definitions	35
2.3.3 Responsibilities of Information Visualization	36
2.3.4 Common Techniques	37
2.3.5 Interaction Techniques	40
2.3.6 Software Toolkits supporting Information Visualization	41
2.3.7 InfoVis Toolkit	41
2.3.8 Prefuse	42
2.3.9 Piccolo	43

2.3.10	Syncfusion Essential Diagram	45
3	PlanningLines	46
3.1	Comparison to PERT and Gantt	46
3.2	Requirements	47
3.3	Design Concept	48
3.3.1	PlanningLine Glyph	49
3.3.2	PlanningLines Display	54
3.4	Example of a Project Plan	57
3.5	Temporal Uncertainties	58
3.6	Discussion	59
4	Prototype Design	61
4.1	Requirements and Environment	61
4.1.1	Basic Environment	62
4.1.2	General Requirements	62
4.1.3	PlanningLines	63
4.1.4	Display	64
4.1.5	Timescale	65
4.1.6	User Interaction Techniques	65
4.2	Toolkits	66
4.2.1	Selection of Toolkits	67
4.2.2	InfoVis Toolkit	69
4.2.3	Prefuse	72
5	Prototype Implementation	76
5.1	Features	76
5.1.1	Data Sources	76
5.1.2	TimeScale	77
5.1.3	Activities of a Plan	77
5.1.4	View	78
5.1.5	Screenshots	79
5.2	Manual	81
5.3	Not Implemented Features	82
6	Prototype Architecture	84
6.1	General	85
6.1.1	Package Structure	85
6.1.2	Coordinate Systems	86
6.1.3	Notes on Prefuse	87
6.1.4	Interfaces	88
6.2	Data	89
6.2.1	Reading Raw Data	89
6.2.2	Data Preprocessing	90

6.2.3	Data Table	90
6.2.4	Reading Data Table	90
6.2.5	PlanningTree	90
6.3	TimeScale	92
6.3.1	General	93
6.3.2	Granularity	94
6.3.3	Graphical Representation	95
6.3.4	TimeScaleLayer	97
6.3.5	Open Problems	98
6.4	Visual Structures	98
6.4.1	Filtering the Tree	99
6.4.2	Visual Structures in ItemRegistry	101
6.4.3	Rendering of Visual Structures	103
6.4.4	View	105
6.5	Graphical Concepts	109
6.5.1	Double Buffering	109
6.5.2	Affine Transformation	110
7	Conclusion	113
7.1	Summary	113
7.2	Evolution	114
7.3	Learned Lessons	115
8	Future Work	116
A	Indeterminacies Calculation Table	117
B	UML Notation	119
	Bibliography	121

Acknowledgements

Chapter 1

Introduction

“If a picture is not worth 1000 words, to hell with it”
-Ad Reinhardt

1.1 Motivation

Planning plays an important role in our life in general. Everyday we have to make a lot of decisions and some of them have to be planned too. To reach a planned occasion or event, other decisions and predictions have to be met. The planning of bigger endeavors can get very complex, therefore we use several techniques to support it. Depending on the context, such techniques reach from simple notices or small graphics to highly sophisticated methods like diagrams or graphical notations. Especially, bigger endeavors or projects requires special attention to planning.

Project Management is a scientific discipline that is concerned with planning and executing business projects. Business projects typically have to organize a lot of events and tasks, manage different resources, and consider a budget. Therefore, several methods and techniques are provided to support the creation and management of plans.

Typically, projects are seen as a sequence of different tasks and activities that are executed along time. Mostly, several resources are assigned to tasks. A typical project plan is an exact time schedule where all tasks and activities are accommodated. To make such a schedule manageable, tasks are very often hierarchically decomposed into smaller pieces of work.

Another discipline that is involved with planning is *Protocol Based Care*. The main aims of it are the creation and spread of medical treatment plans. A medical treatment plan is a recommendation for treating diseases or performing therapies that is used by clinical staff. Such a plan typically has to consider different states of patients. Mostly, the execution of the plan has to react on patients' state, therefore, such plans typically work with conditions. Similar to *Project Management*, tasks of a treatment plan also refer to time.

Creating a time schedule for any kind of plan can be a real challenge. Often the success of the whole project depends on it, therefore, special attention has to be paid towards this issue. Schedules are often created by assuming fixed durations of tasks. But as such assumptions are only predictions of the future they may not correlate with reality. Therefore, some flexibility is claimed in plans.

PlanningLines is a visualization concept developed for the representation of *Asbru* plans. *Asbru* is a formal representation language for medical treatment plans. It provides a mechanism that enables the creation of flexible plans by considering temporal uncertainties. This is obtained by using a set of time attributes that assigns intervals instead of fixed start and end-points and two different durations on how long an activity can last. Therefore, the visualization technique *PlanningLines* also supports the flexible creation of plans.

Generally, visualization of plans is a common technique to support planning. The scientific discipline *Information Visualization* provides methods and concepts which support the representation of information visually. The base of every *Information Visualization* is an underlying notation on how information is communicated to users. Even if the notation of *PlanningLines* originally was developed to visualize *Asbru* plans, the concept can also be applied to *Project Management* plans. Strictly speaking, *PlanningLines* was developed by considering two common visualization techniques of *Project Management*: PERT and Gantt.

The aim of this thesis is the implementation of a prototype that applies *PlanningLines* to *MS-Project* plans as well as to *Asbru* plans. The resulting *Information Visualization* should demonstrate a way of treating temporal uncertainties in plans visually.

Thus, the prototype should also help to prove the concept of *PlanningLines* regarding its usability in a real application. Furthermore, the prototype should prove that *PlanningLines* works with common *Project Management* plans too. The concept of *PlanningLines* should extend the original concept by several techniques known of the domain of *Information Visualization*. Another aim of this thesis is the evaluation of open-source toolkits for graphical representation regarding their operability.

1.2 Overview of the Thesis

This thesis starts with a State of the Art report (Chapter 2) that outlines all surrounding scientific disciplines the prototype is concerned with. Besides discussing *Project Management* and *Protocol Based Care*, *Information Visualization* is described in detail. Of each discipline an overview is given, further some details regarding to proper work are described in detail.

Chapter 3 introduces the concept of *PlanningLines* in detail. The whole

notation is explained, and special cases of *PlanningLines* are outlined.

After the theoretical part of this thesis, the practical part starts by listening the requirements of the prototype (Chapter 4). There, also the selection of adequate toolkits is described. Chosen toolkits are outlined in depth.

Chapter 5 gives an overview of all implemented features of the prototype. Furthermore, possible extensions of the functionality for future work are given here. Chapter 6 gives insights into the architecture of the prototype. Also some technical programming concepts used in the *Information Visualization* are described in detail.

Concluding, a recapitulation of the work is given in Chapter 7 and a short outline of future work in Chapter 8.

Chapter 2

State of the Art

This state of the art Section tries to describe the related scientific environment the prototype is confronted with. Especially *Project Management* and *Information Visualization* are very large and complex disciplines with countless different approaches and techniques developed over decades. *Protocol Based Care* is a younger discipline, but also here a lot of different projects researching new ways in healthcare can be found. Therefore, a brief overview of each discipline is given and related matters are described in depth.

2.1 Project Management

2.1.1 Overview

“Project Management, in its modern form, began to take root only a few decades ago. Starting in the early 1960s, businesses and other organizations began to see the benefits of organizing work around projects and to understand the critical need to communicate and integrate work across multiple departments and professions” [Sisk, 1998]. Therefore, *Project Management* has established itself as an own international discipline and “project manager” has become an own profession in the past years.

Optimizing resources and preparing a time schedule are two of the most important parts in *Project Management*. One of the pioneers in scientific management was Frederick Winslow Taylor (1856 - 1915) who researched methods to improve industrial efficiency. His main focus was set to analyzing work and its elementary parts (e. g. , movements, proportions of shovels) with the aim to improve productivity [Wikipedia, the free encyclopedia, 2006a].

Taylor's associate, Henry Laurence Gantt (1861-1919), studied the order of work and developed Gantt charts to illustrate workflows (first published 1910 in “The Engineering Magazine”, NY). These charts are such powerful analytical instruments in management that they are used nearly unchanged in modern *Project Management* [Sisk, 1998].

Because of large and complex plans of governments and their institutions in World War II and in the Cold War the term project was coined (e. g. , Manhattan Project). New instruments like network diagrams (Program Evaluation and Review Technique (PERT)) and the Critical Path Method (CPM) were introduced to speed up military projects in the USA. The process flow and structure of military undertakings quickly spread into many private enterprises and organizations. So, and the term *Project Management* arose [Wikipedia, the free encyclopedia, 2006c].

Today, *Project Management* is one of the most important instruments in management and gains importance because of sharing work within businesses. In a modern business it is essential to work together on common goals to survive. Especially large cooperations are spread in different locations and work (own or outsourced) has to be coordinated. But even small businesses are dependent on good management to produce high quality and affordable products.

2.1.2 Definitions

Before discussing *Project Management* itself, some common definitions are necessary.

Process

A process is a sequence of activities that takes a resource and produces (transforms the input) an outcome [Harrington, 1991]:

“Any activity or group of activities that takes an input, adds value to it, and provides an output to an internal or external customer.”

Both, the input and the output of a process, can be nearly everything like time, space, expertise, or any other resource. In general, a process takes up some time and it may be categorized as singular, recurrent, or periodic process. It is important to note that every process must produce a value for its customer - otherwise it should be eliminated because of wasting resources.

An activity stands for any possible task like mechanical or creative work, research, or another process (sub process).

Project

There are a lot of different definitions of the term project depending on context and scientific discipline. With regards to Section 2.1.2 a project can be seen as a set of different processes that have to be performed to reach the end of a project.

Generally, a project stands for a bigger endeavor, but in today's usage the term often stands for any work being done by more than one person - even if it is the normal operational work of an organization. This usage is inadequate since one of the most important characteristic of a project is its unique result. That means that a project is something new for the executive organization or involved people.

One of the most common definitions for a project is [Wysocki et al., 2000]:

“A project is a temporary sequence of unique, complex, and connected activities having one goal or purpose and that must be completed by a specific time, within budget, and according to specification.”

Another useful definition of the Project Management Institute (PMI) [Project Management Institute, 2000]:

“A project is a temporary endeavor undertaken to create a unique project or service.”

According to these definitions, a project is temporally limited, that means it has a clearly defined beginning and ending. Another important characteristic is the complexity - many different activities (often performed by various organizations, departments, or people) have to be identified, planned, and organized. Due to the usual interdependencies between activities it is necessary to plan the logical and chronological order of the different tasks exactly.

2.1.3 Responsibilities of Project Management

As projects can be classified into types like complex tasks, temporary organizations, or social systems and can be differentiated by industry, ownership, duration, and so on, there is no “one way” in management of these [pma - Project Management Austria, 2002, Gareis, 2000]. Of course *Project Management* follows some general paradigms and a lot of methods and techniques are well established in this discipline. But managing a project is a task with a lot of different aspects and success often depends on personal skills and experience of the managers.

Project Management is a combination of many scientific disciplines like General Management, Risk Management, Quality Management, and - depending on the ownership - technical skills like Software Engineering, Medical Treatment etc. Therefore, a project manager has to cover a lot of different knowledge in combination with social and technical skills which makes *Project Management* a scientific discipline of its own.

A project (see Section 2.1.2) is typically a large, new, and complex mission to handle. Therefore it has to be analyzed, planned, and prepared carefully long time before the ultimate start. Otherwise it is endangered to fail, or at least involves large costs and a lot of time.

The initiating and planning-phase helps the management to determine resources, define subprocesses and milestones, and get an overview of the amount of work that must be done to meet the requirements. But also expected costs, risks, problems and their solutions can be cleared up beforehand [Project Management Institute, 2000]:

“Project management is the application of knowledge, skills, tools, and techniques to project activities to meet project requirements. Project management is accomplished through the use of the processes such as: initiating, planning, executing, controlling, and closing. The project team manages the work of the projects, and the work typically involves:”

- *“Competing demands for: scope, time, cost, risk, and quality.”*
- *“Stakeholders with differing needs and expectations.”*
- *“Identified requirements.”*

According to this definition, five different phases can be differentiated:

Initiating-Phase

Before the planning-phase starts, goals and objectives must be defined. Sometimes this task is not so difficult (e.g., building a bridge) but very often finding the specifications is a long process (e.g., large IT-projects) that should be handled exactly and precisely (changing the requirements at a projects' runtime is not effortless or even impossible). Furthermore, involved people or businesses, expected costs, and resources must be identified or elected.

Planning-Phase

Generally, planning means to make predictions for the future. Activities have to be planned and prepared, interdependencies identified, teams selected, etc.. All these tasks have to be organized regarding to their logical and temporal prerequisites. As well, eventual upcoming problems should be recognized and treated in this phase, otherwise they could result in unexpected complete re-planning while execution.

This leads to several problems, especially with regards to temporal sequences. Often it is not possible to determine the durations of activities

exactly - assumptions and estimations have to be considered within a plan. These assumptions lead to temporal uncertainties within a plan which have to be treated. At the end of this phase, a formal and detailed project plan should exist.

Execution- and Controlling-Phase

During the “lifetime” of a project, the main task of *Project Management* is to control the work. A common method for controlling the progress of a project is the concept of milestones (verification of in-time completion of certain sub aims, defined in the project plan (see Section 2.1.4). Whenever a task does not proceed as expected, responsibility for managing the further execution lies in *Project Management*.

Closing-Phase

Closing a project typically starts some time before the work is done. Depending on the definitions of the initial and planning-phase, final tests, presentations, and acceptance tests fall into this part of *Project Management*. But, also reviews and discussions about the evolution are appropriate instruments to analyze the work being done and get new experience for further projects.

Interaction of Phases

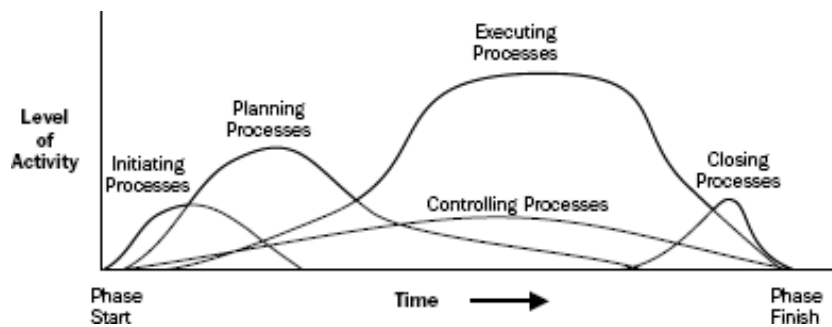


Figure 2.1: Sample of overlapping project phases listened along the time [Project Management Institute, 2000].

As everything in *Project Management*, the phases differ from project to project because of their own characteristics (e.g., ownership or amount). Generally, it is not possible to assign work to a certain phase (e.g., in an IT-project defining the system architecture can be done in the planning or execution-phase) or estimating the amount of a certain phase. Moreover, there are not many hard rules defined describing how a project has to be

managed - just suggestions and some common paradigms. With respect to time, the different phases often overlap. Such intersections can be found especially in really large projects where first parts are already finished while the whole project is still in the planning-phase.

2.1.4 Problems in Project Planning

Although, there are a lot of scientific techniques and concepts for managing large projects, statistics show that many projects fail or do not meet the requirements in practice (see Figure 2.2 that shows a statistical evaluation of 30000 IT projects).

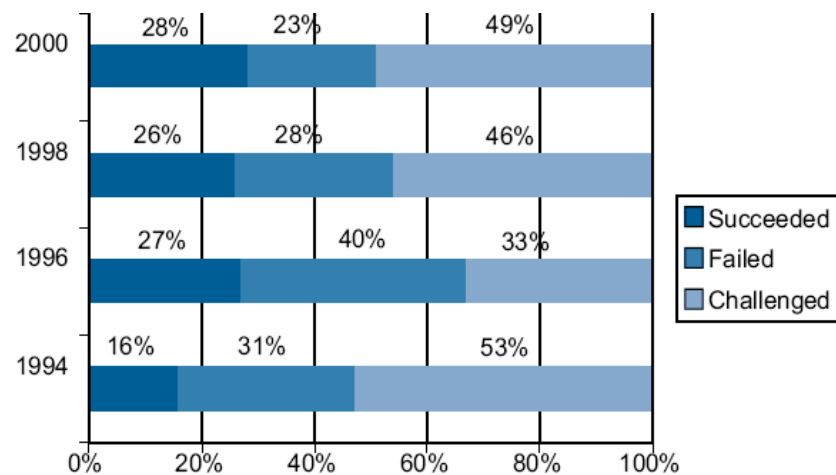


Figure 2.2: Project resolution history (1994-2000) [The Standish Group, 2001].

The Standish Group categorizes IT projects into three different resolution types [The Standish Group, 2001]:

- **Successful:** The project is completed on time and on budget, with all features and functions originally specified.
- **Challenged:** The project is completed and operational, but over budget, over the time estimate, and with fewer features.
- **Failed:** The project is canceled before completion or never implemented.

Meeting the statement “*The project success depends on the relationships of the project to the relevant project environments*” [Gareis, 2000] is a quite difficult task for the management. There are countless reasons that can make a project fail like unexperienced project managers, competence problems within the team, missing technical skills, or inaccurate planning [The

Standish Group, 2001]. Wherever people work, mistakes are made. It is not possible to consider each eventual incident, but especially in the planning-phase a lot of mistakes can be prevented.

Project Plan

The major goal of the planning-phase is an exact and accurate project plan, i. e. , a document describing all single steps that have to be performed to meet the requirements. Besides formal descriptions of the project, the project plan also contains detailed time schedules, personal responsibilities and liabilities, and cost and resource-planning.

To get an overview of the whole project, it is necessary to divide the work into some clear and logical tasks. Many upcoming activities and tasks are already evaluated (i. e. , done in preceding projects) - others have to be defined and worked out by the involved people of the project. Some activities are well known (e. g. , common operational works of the organization), some will be outsourced, and others will have to be organized just from the beginning. Complexity, existing knowledge, amount of involved people, and needed resources of a task determine how exact and detailed it has to be planned.

Once all tasks are specified and worked out, a project plan can be created. All the tasks and their subtasks must be organized according to their logical and chronological order. Depending on the complexity, this can be quite difficult work. Several planning techniques (like Gantt or PERT) and software tools (see sections 2.1.6) implementing these techniques can help to represent all coherences visually. With corresponding tools, different views (charting techniques) and levels of detail can be created and displayed. Typically, it is necessary to apply several charting techniques as they address different objectives (see Section 2.1.5).

Temporal Uncertainties

Especially a correct time scheduling is a decisive but sensible factor within the planning-phase of a project. “*Time is money*” is a significant slogan in economy, but arranging a timetable is a very difficult task for the management of a project. For reasons of expense, the needed time of activities or tasks required by a project is often calculated under the assumption of ideal conditions and circumstances. Every unexpected event can result in an inaccurate time schedule for a single activity and this again can result in a collapse of the entire project. Anyway, management cannot consider each eventuality within the time schedule, some temporal uncertainties will always remain. Good *Project Management* is distinguished by a flexible and realistic handling of chronological sequences.

With realistic assumptions about durations of tasks, adequate methods,

and software tools, an adequate and efficient timetable can be produced. This means, that each single task gets enough time for fulfilling the associated requirements. Estimating the time for each task should be done together with the executive and responsible people. To handle eventually upcoming delays, several models can be found. In general every model depends on one of the following three approaches [Vidal, 2004]:

- *Reactive*: The project plan is created without consideration of temporal uncertainties and the proper execution of the plan starts as usual [Smith, 1994]. If there is a violation of a time constraint, a possible solution to pass the situation will be searched or, if not possible, a complete re-planning will be done from this moment on. The advantage of this technique is, that a lot of possible uncertainties can be ignored while execution runs as expected.
- *Progressive*: Planning is done until the first expected temporal uncertainty is reached. Whenever this point is reached, the next Section of the plan will be created. These steps are continued again and again, until the project is finished. This technique is often referred to as a rolling time-horizon plan generation [Vidal et al., 1996] and is a good approach in smaller projects. Once there are a lot of parallel or sequential processes within a project this technique is not practicable.
- *Proactive*: This approach tries to consider each possible temporal uncertainty before it actually occurs. Applying this way requires a lot of information about every activity within the project. One idea is to calculate the execution with fuzzy durations and compute the probability of success to fulfill the project. Another idea is to add flexibility to the plan by using “floating” begins and ends (see Figure 2.3).

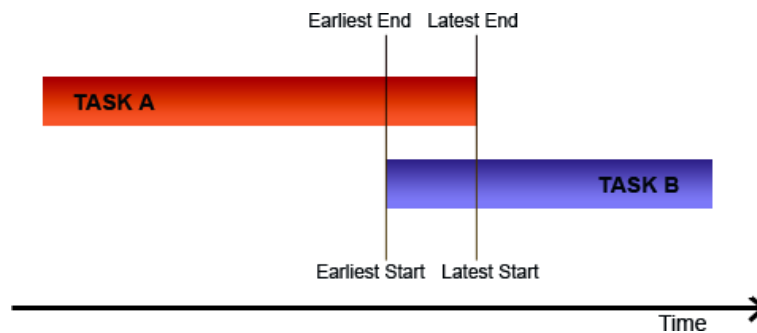


Figure 2.3: Sample of “floating” begins and ends.

Of course, thinking about temporal uncertainties does not guarantee a smooth flow of the project, but it is more probable. Especially the *proactive*

approach provides possibilities that smaller delays of an activity do not affect the entire project. But also shorter durations of tasks than expected can influence the progress in a positive way. Anyway, in case of a failure, *Project Management* can still continue with the *reactive* technique to find the way back to a working plan.

Often, common planning techniques are extended by possibilities of treating temporal uncertainties. A relatively new charting technique is *Planning-Lines* (see Section 2.1.5) which already treats temporal uncertainties in a *proactive* manner.

Treating temporal uncertainties costs time within the planning-phase, but, besides the mentioned advantages, there is also another benefit that should legitimate this work: The necessary analyses to create a detailed time schedule result in additional information about activities. The gain of information flows back into the management of a project. On the one hand, problems may be identified in the planning-phase and can be treated beforehand. On the other hand, expected slack times can be minimized as detailed information about the task itself and surrounding tasks are available.

Typically, the results of creating a time schedule and treating temporal uncertainties improve the chance to complete single activities at all and in time.

2.1.5 Graphical Plan Representation - Charting Techniques

One of the most important methods to support organizing and planning in *Project Management* is the graphical representation of projects. On the one hand, project managers can get an overview of the project or parts of it (today, plans are usually hierarchically decomposed to view them with different levels of detail). On the other hand, plans help to control the progress of the entire project. There are a lot of different techniques, the most common are described below considering an example project.

Each technique follows its own objectives, therefore in *Project Management* more different techniques are applied to a project typically. Of course it is possible to create planning charts on paper, but today there are a lot of computer programs to do this work. Software provides easy ways to realize the exact charting of plans, re-planning or modifications can be done easily. This implies that realistic controlling can be performed during “runtime” of a project.

Work Breakdown Structure

Work Breakdown Structure (WBS) is a common planning technique used many organizations like National Aeronautics and Space Administration (NASA) [NASA, 1994]:

“The purpose of a Work Breakdown Structure (WBS) is to divide the program/project into manageable pieces of work to facilitate planning and control of cost, schedule and technical content. A WBS is written early in program/project development. It identifies the total work to be performed and divides the work into manageable elements, with increasing levels of detail.”

This technique is applied at the beginning of the planning-phase. An accurately done WBS allows the project team to get an exact reflection of the whole project. Usually, ends of critical elements within the WBS are defined as milestones, a useful technique helping *Project Management* to acquire the progress of a project in the controlling-phase. The most common method to represent a WBS is a hierarchical tree (see Figure 2.4) together with a describing table containing detailed information. It is important to notice that a WBS does not represent any sequence or scheduling of the project but it provides an exact view of hierarchical decompositions.

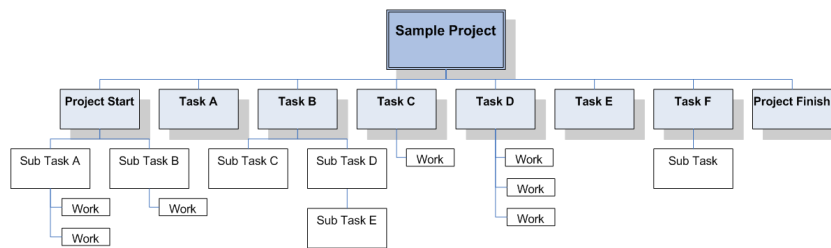


Figure 2.4: Sample of a Work Breakdown Structure.

Gantt Charts

The main task of Gantt charts is to show the timing of tasks or activities in a project [Morris, 2000]:

“A chart that depicts progress in relation to time, often used in planning and tracking a project.”

All tasks are listed along the vertical axis, time along the horizontal axis. A single task is represented by a rectangular bar with a fixed height and a width depending on the duration of a task. The horizontal position refers to the timescale along the horizontal axis (see Figure 2.5). A Gantt chart in its original meaning provides an easy way to represent temporal facts about a project, but it is not possible to depict dependencies or hierarchical decompositions between tasks.

Therefore Gantt charts were enhanced by several variants and combinations of variants over the years. One common and useful extension is to

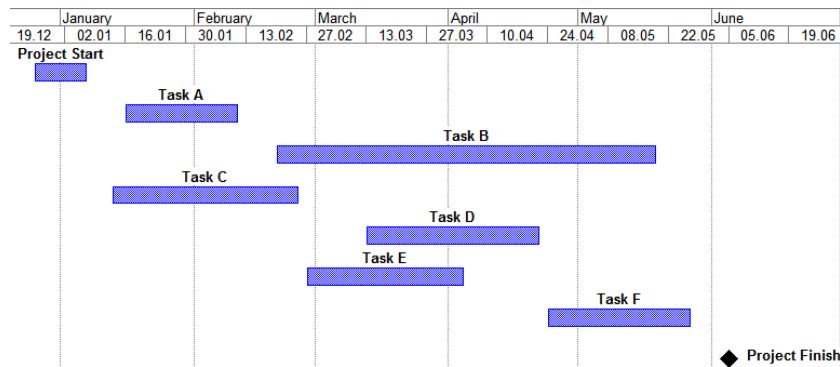


Figure 2.5: Sample of a Gantt chart.

display interdependencies of tasks. To realize this demand, Gantt charts are seen as a network of tasks, logical relations are represented with arrows between activities (see Figure 2.6). Due to this combination with a network technique, Gantt charts are a much more powerful instrument to plan complex projects.

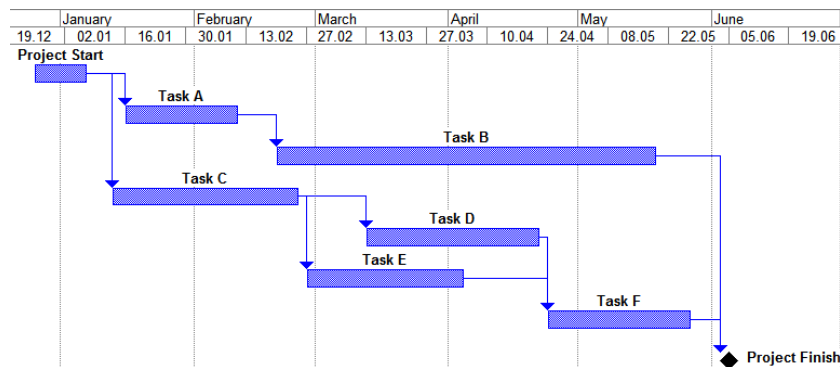


Figure 2.6: Sample of a Gantt chart with logical relations.

Another common extension is the representation of hierarchical tasks. As usual in a WBS, complex tasks are divided into smaller subtasks. To represent such composite tasks special bars with triangular ends are used (see Figure 2.7). This method allows the management to split a bigger project into single, more detailed charts.

PERT

The Program Evaluation and Review Technique (PERT) depicts information about tasks, durations, and dependencies between activities of a project.

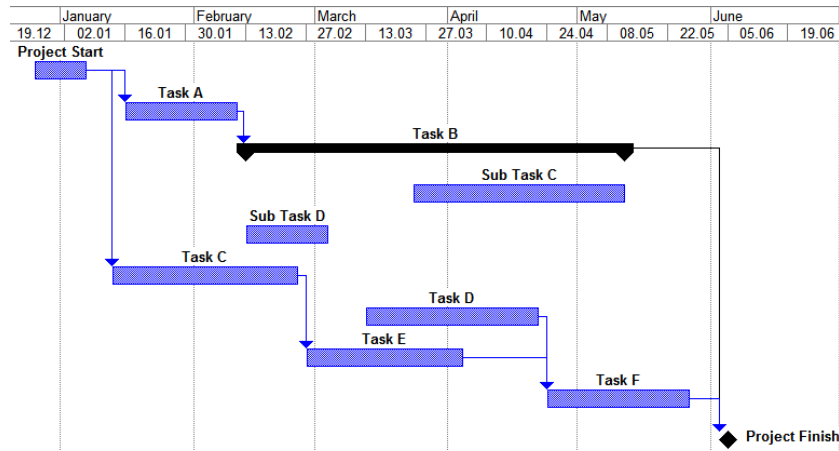


Figure 2.7: Sample of an expanded hierarchical Gantt chart.

While a WBS represents only tasks, PERT also shows the chronological sequence of them. Therefore, PERT provides more details of a project and is typically applied later in the planning-phase than other techniques. Although it is possible to read out dependencies of tasks, the main focus of PERT is set to the possibility of temporally analyzing the project (duration of subtasks, entire project, etc.).

A PERT chart is a network of all tasks within a project. There are several ways for representing a task within PERT charts, but most common are boxes with the task name and information about the most important time factors (i.e., Earliest Starting Time (EST), Latest Starting Time (LST), Earliest Finishing Time (EFT) and Latest Finishing Time (LFT)) (see Figure 2.8).

Early Start	Duration	Early Finish	Task Name	
Task Name			Scheduled Start	Scheduled Finish
Late Start	Slack	Late Finish	Actual Start	Actual Finish

Figure 2.8: Sample of different PERT notations.

Each task is connected with its successor tasks (logical relations). Usually, this relationship means, that the prior task must be finished before the later one can start. PERT does not work with a timescale, therefore inaccurate scheduling is not as obvious as in Gantt charts. But generally, PERT charts provide much more information about temporal factors and conditions than simple Gantt charts or other network plans. Additionally, there are several analytical ways available to explore problems or critical paths within a project.

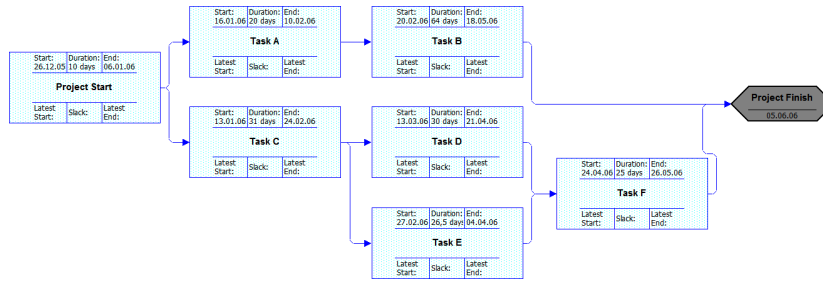


Figure 2.9: Sample of a PERT chart.

Besides the representation, PERT also provides an analytical technique to determine durations of single tasks or the whole project [Modell, 1996]. This technique is based on three assumed values: optimistic duration (o), pessimistic duration (p), and most likely duration (m). With a simple formula the expected duration (e) is calculated:

$$e = \frac{o + 4 * m + p}{6}$$

Another way to deal with temporal uncertainties is to determine the following four quantities: *Earliest Start*, *Latest Start*, *Earliest End* and *Latest End*. With this additional declarations it is possible to calculate the critical path of a project.

PlanningLines

Besides the mentioned widespread planning techniques, there are a lot of others aiming at special problems in *Project Management*. A relatively new one are *PlanningLines*, developed as a result of a user study with physicians (see Section 2.2.4) by the Institute of Software Technology and Interactive Systems, Vienna University of Technology [Aigner, 2003]. Originally, this visualization technique was intended to represent medical treatment plans, but it can also be applied to *Project Management*. *PlanningLines*' primary attention is dealing with temporal uncertainties (see Section 2.1.4) in a *proactive* way.

PlanningLines can be seen as a combination of Gantt and PERT charts. Instead of single rectangular bars, as used in Gantt charts, complex glyphs (single *PlanningLines*) are applied to each task. Such a glyph represents the typical data of a common PERT task visually (see Figure 2.10). This glyph allows to display the most important temporal attributes clearly.

A glyph consists of the following elements [Aigner et al., 2005a]:

- Start-interval (Earliest Starting Time (EST) and Latest Starting Time (LST));

- Minimum Duration (minDu) and Maximum Duration (maxDu); and
- End-interval (Earliest Finishing Time (EFT) and Latest Finishing Time (LFT)).

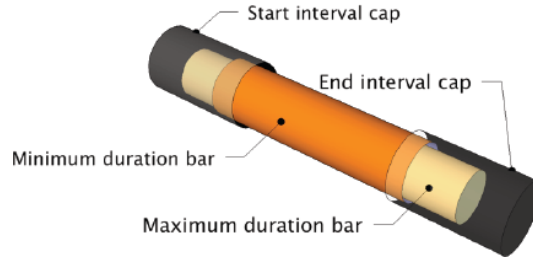


Figure 2.10: Concept of a PlanningLine glyph.

The duration bars are encapsulated between the bounding start and end-intervals. As the caps stand for intervals, the actual start or end must be within the respective cap, and the actual duration must be between minDu and maxDu (for more informations see Section 3.3.1).

Similar to Gantt charts, *PlanningLines* are arranged along the vertical axis of a chart, referring to a timescale on the horizontal axis (see Figure 2.11). This representation provides the management of a project to take advantage of the clarity of Gantt charts in combination with the detailed temporal view of PERT charts.

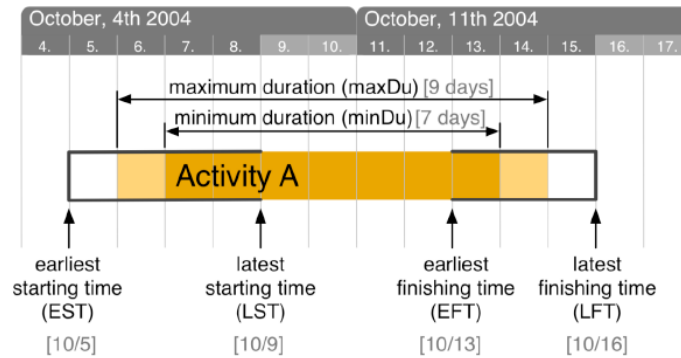


Figure 2.11: Sample of a PlanningLine regarding to a timescale.

As mentioned before, Gantt charts are available in many different variations, like hierarchical views and relationships between activities. The only difference of *PlanningLines* in comparison with Gantt is the visual representation of tasks. Therefore, most of the common enhancements of Gantt charts can be applied to *PlanningLines* too, which makes them a powerful alternative in *Project Management*.

2.1.6 Software Tools

As usual in nearly every discipline, many different software tools are in use. Software that facilitate *Project Management* typically support planning and organizing of budget, time, and resources. Mostly, data is organized in tables and the results are communicated visually. *Project Management* tools can be differentiated into desktop and web applications, as well as free and proprietary implementations. Especially, tools that provide charting techniques are of particular interest.

MS-Project

Microsoft Project (see Figure 2.12) is an additional part of the MS-Office package, a complete solution of standard business software. Because MS-Office is widely used worldwide and the cooperation between the applications, *MS-Project* is the world leading software tool used in *Project Management* [Microsoft, 2006].

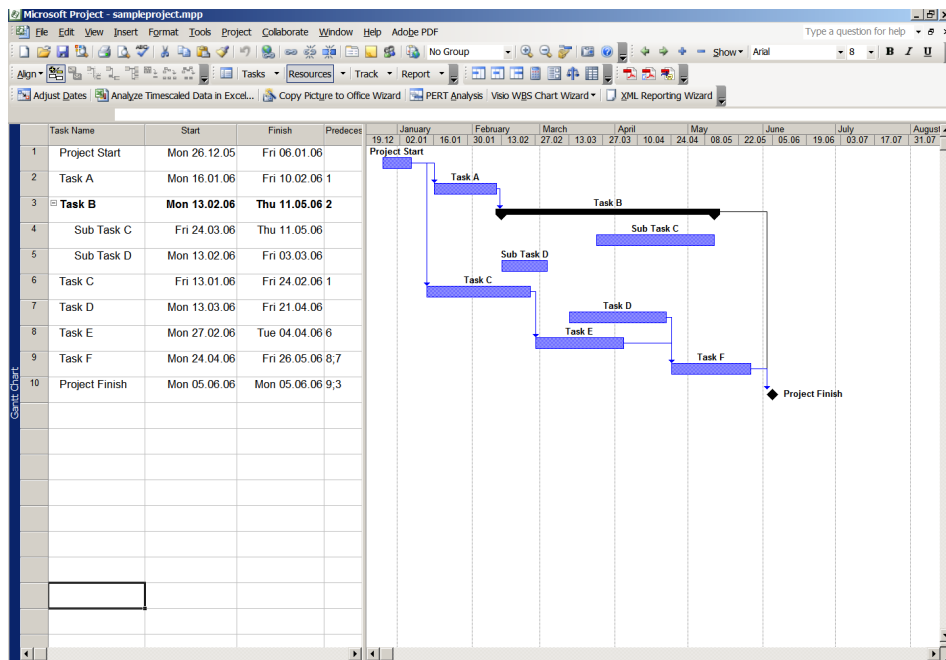


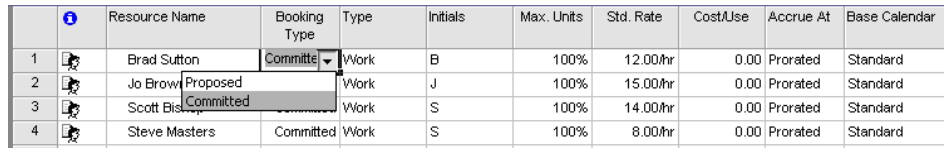
Figure 2.12: Screenshot of MS-Project showing a Gantt chart.

MS-Project (in combination with other MS-Office tools) is designed to meet many different aspects of *Project Management*:

- To support planning, different visualization techniques can be applied (variations of Gantt and some kinds of network plans like PERT). Once

one chart is created, other basic views are produced automatically (see figures 2.5 - 2.9). Generally, all techniques are extended by the possibility of hierarchical decomposition, this allows to work with different levels of detail.

- A detailed resource planning is realized by a groupable data table (see Figure 2.13). Once a resource is defined, it can be applied to any task of any project (shared resource pool). This feature facilitates usage of *MS-Project* with more different projects depending on the same resources.
- During execution, it provides methods like tracking the progress of a project, scheduling time, managing the budget, and creating and sending reports to concerned people automatically.



		Resource Name	Booking Type	Type	Initials	Max. Units	Std. Rate	Cost/Use	Accrue At	Base Calendar
1		Brad Sutton	Committed	Work	B	100%	12.00/hr	0.00	Prorated	Standard
2		Jo Brown	Proposed	Work	J	100%	15.00/hr	0.00	Prorated	Standard
3		Scott Biss	Committed	Work	S	100%	14.00/hr	0.00	Prorated	Standard
4		Steve Masters	Committed	Work	S	100%	8.00/hr	0.00	Prorated	Standard

Figure 2.13: Sample of resource planning table in MS-Project.

Analytical methods are applied automatically by *MS-Project* during the planning-phase and during execution, values are updated periodically or on demand. These include CPM and PERT calculations (see Section 2.1.5), but also the budget is calculated by considering specific costs for each needed time unit of resources.

Another advantage of *MS-Project* is its compatibility to MS-Visio, a charting tool. Each project can be exported to MS-Visio as a WBS. Based on this WBS, further special charting techniques can be applied (e.g., charting with special symbols for presentations, etc.).

Generally, *MS-Project* is a desktop tool for managers, but in the meanwhile, professional version assumed, also centralized tasks and automated processes can be managed globally, using the *MS-Project* server in combination with web-access. This extensions have caused particular interest in this product, especially of large and widely spread companies.

GanttProject

GanttProject (see Figure 2.14) is an open-source solution written in Java hosted by SourceForge.net [SourceForge, 2006] and is available for free. Besides an own file format, GanttProject supports *MS-Project* files too.

GanttProject is designed similar to *MS-Project*, but provides not that much functionality. It is restricted to the basics of Gantt and PERT charts,

CPM is applied as analytical method, a calendar is included, and it also provides limited resource-planning.

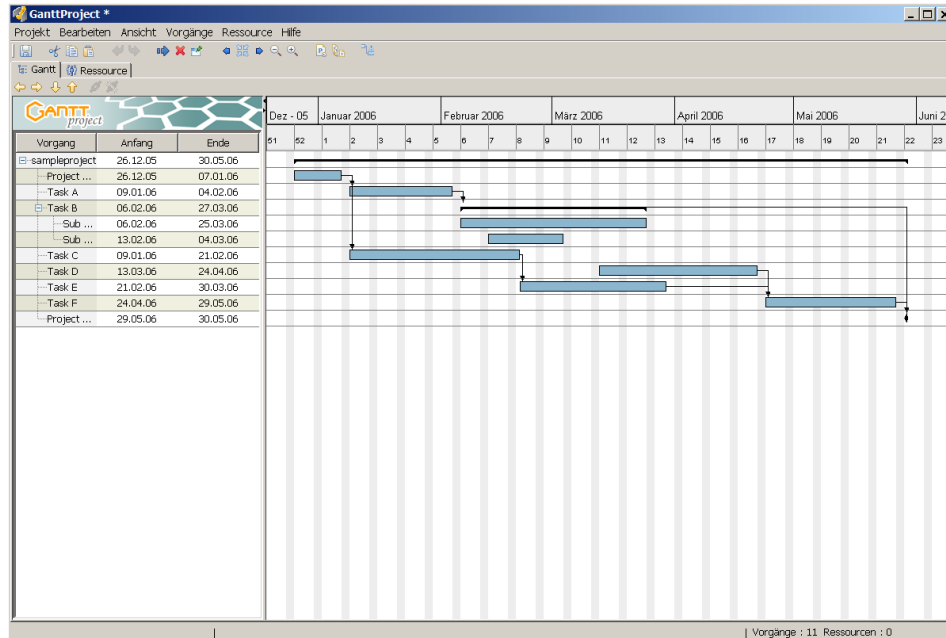


Figure 2.14: Screenshot of GanttProject showing a Gantt chart.

Concluding, GanttProject is not adequate to manage large business projects, but it is suitable for planning small projects.

@Task

@Task is a proprietary *Project Management* solution for large enterprises [@Task, 2006]. Data of projects are stored centralized in a database, content is provided by servers cross the web using several technologies (HyperText Markup Language (HTML), Java, Wireless Application Protocol (WAP), etc., see Figure 2.15). This ensures a permanent and independent access to managed projects, but also allows authorized users to manage the software itself. @Task interacts with *MS-Project* and other third party applications. Scalability is reached by the possibility of running the application on several clusters.

@Task provides a lot of standard features like resource-planning, project-planning (Gantt, PERT, WBS), time scheduling, and some analytical methods (see Figure 2.16). To facilitate collaboration, each user works on a personalized portal (see Figure 2.17), containing all necessary and relevant information, including a personalized calendar that is automatically generated. Besides that, @Task provides various possibilities of communication

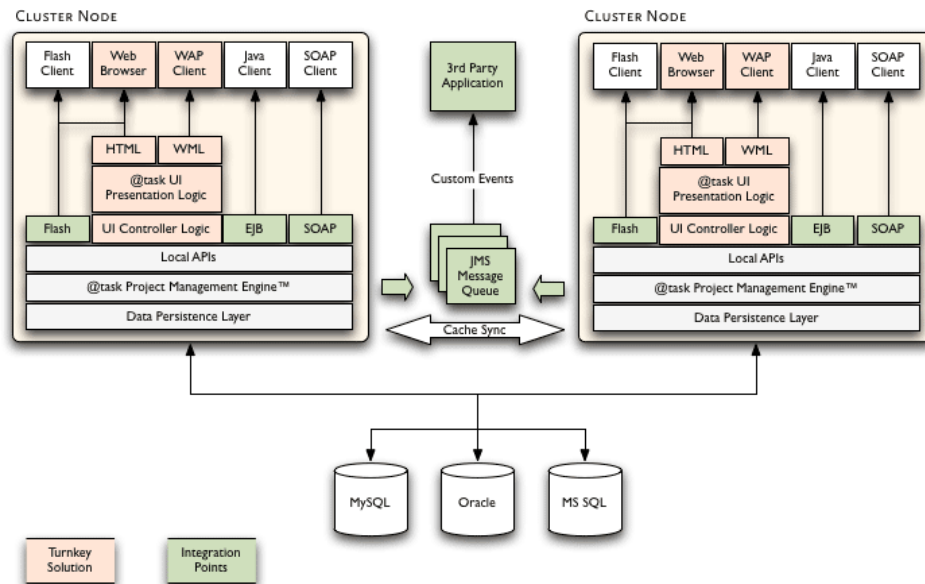


Figure 2.15: System Architecture of @Task [Task, 2006].

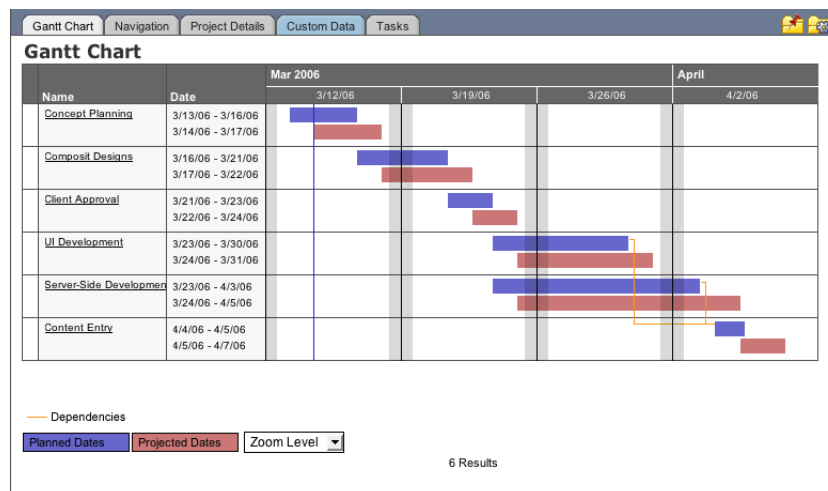


Figure 2.16: Sample of a Gantt chart in @Task [Task, 2006].

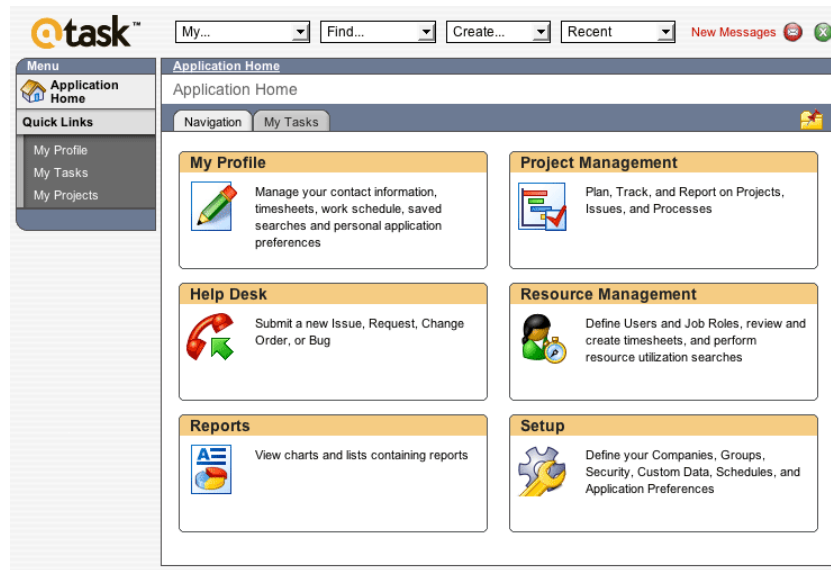


Figure 2.17: Sample a personalized portal in @Task [Task, 2006].

(i.e., sending generated reports periodically to related members, e-mail notification on new project issues, etc.).

After discussing important aspects of *Project Management*, the next Section gives an introduction of *Protocol Based Care*. There are some correlations between these disciplines, especially between project-planning and clinical guidelines which are representing plans for medical treatment.

2.2 ProtocolBased Care

2.2.1 Overview

In our modern world, computers and machines strongly influence our life and every science discipline. Today's health care would not work without all the available technology, but especially in theoretical parts of medicine technical innovations are hardly used. As medical treatment plans are neither standardized nor written down mostly, new ways of communication are demanded in this domain. Especially the demands of more quality and standardization in medical treatment could be solved through applying new techniques of information technology.

There are a lot of definitions of *Protocol Based Care* depending on context and country. Basically, *Protocol Based Care* is a multi-disciplinary approach to provide clear standards and statements in health care. The main aims are safety, quality, and clinical effective ways of treatment. One

way to meet these demands are clinical guidelines [NHS - Modernisation Agency, 2002].

2.2.2 Clinical Guidelines and Protocols

An important part of *Protocol Based Care* is computer support for clinical treatment. Therefore, different organizations have turned their attention to the realization of this. There are several ways to make a computer deal with information, but especially in the medical domains it is not that easy to join all requirements (complexity, constraints, conditions, etc.) of treatment plans.

An approach to meet these demands are computerized clinical guidelines. An important benefit of clinical guidelines is that they can be processed by computers. This makes them easy to spread locally, therefore standardization and quality in medical treatment can rise significantly if they are created properly, tested, and proved [Woolf et al., 1999].

Such a guideline or protocol is an explicit recommendation for clinicians. This can help to proceed the work and clinicians can update their knowledge to the latest scientific facts. To spread clinical guidelines locally, a framework (e.g., internet platform) must be provided to which clinicians are connected [NHS - Modernisation Agency, 2002].

A common definition of clinical guidelines is [Shahar, 2002]:

“Clinical guidelines are a set of schematic plans, at varying levels of abstraction and detail, for management of patients who have a particular clinical condition (e.g., insulin-dependent diabetes).”

“Clinical protocols are typically highly detailed guidelines, often used in areas such as oncology and experimental clinical trials.”

A clinical guideline can be seen as a reusable skeletal plan to identify or treat a particular medical condition. But of course applied treatment differs from patient to patient because of respective characteristics. Therefore, a guideline needs room for some flexibility to ensure an adequate treatment considering all circumstances. Another very important issue is the possibility to compose plans by using predefined plans (or sub plans). It does not make sense to define each single step of a treatment plan newly whenever it is needed, a better way is to take care of reusing elements (i.e., hold them in a database or library). That allows composing new plans with already defined elements, saving time, and avoiding mistakes.

To meet these requirements, it is important to care about an effective way of representation. One useful way to map such a plan is a formal representation language. Although defining such a representation language is a lot of work, but, once developed, there are some advantages compared to other methods known in knowledge management. Because a language

has its own clear syntax, logic, and semantic, it can be used by different organizations to build their own applications upon such a representation language. It can be seen as an standardized and expandable base in a domain that provides the possibility to develop each preferred application.

Conceivable applications upon a formal representation language are execution units (local on a computer, but also client-server architectures or internet solutions), graphical visualizations of plans, and tools to create or edit a plan (architecture tools). Even if automatic execution of treatment plans by computers is imaginable (and desirable, but this is controversial [Woolf et al., 1999]), today, a lot of attention is drawn to find adequate methods to represent treatment plans. Additionally, supporting graphical visualizations of protocols is an important issue researchers are concerned with.

Computerized treatment plans themselves are generated by using formal methods of the domain of knowledge management. Therefore, ways have to be found to communicate such guidelines to clinical staff who should finally work with them. Besides generating textual plans, graphical representation of treatment plans is a possibility to impart a guideline to clinicians [Aigner and Miksch, 2004].

2.2.3 The Guideline Representation Language Asbru

One formal representation language to create clinical guidelines is *Asbru*, developed by the Institute of Software Technology and Interactive Systems, Vienna University of Technology and the Section on Medical Informatics, Stanford University.

Asbru is a time-oriented, intention-based, skeletal plan-specification representation language that is used in the *Asgaard Project*¹ to represent clinical guidelines and protocols in XML. *Asbru* can be used to express clinical protocols as skeletal plans [Friedland and Iwasaki, 1985] that can be instantiated for every patient (for an example see Figure 2.18). It was designed specific to the set of planmanagement tasks [Miksch, 1999]. *Asbru* enables the designer to represent both the prescribed actions of a skeletal plan and the knowledge roles required by the various problem-solving methods performing the intertwined supporting subtasks. The major features of *Asbru* are that

- prescribed actions and states can be continuous;
- intentions, conditions, and world states are temporal patterns;
- uncertainty in both temporal scopes and parameters can be flexibly expressed by bounding intervals;

¹In Norse mythology, Asgaard was the home of the gods. It was located in the heavens and was accessible only over the rainbow bridge, called Asbru (or Bifrost) (For more information about the Asgaard project see <http://www.asgaard.tuwien.ac.at>).

- plans might be executed in sequence, all plans or some plans in parallel, all plans or some plans in a particular order or unordered, or periodically;
- particular conditions are defined to monitor the plan's execution; and
- explicit intentions and preferences can be stated for each plan separately.

A clinical guideline represented with *Asbru* is a set of plans and actions that are executed in a defined sequence. To control the execution of a plan several conditions can be set like filter precondition, abort condition, and complete condition. The execution sequence of actions (variable assignment, forced user inputs, or conditions) or new plan-activations (plan, user performed plan, or cyclic plan) are defined in the plan-body (see Figure 2.18) of each plan [Aigner and Miksch, 2004]:

- *sequential*: The given actions are executed in the given order.
- *parallel*: The given actions are initialized at the begin and executed parallel.
- *any-order*: Same as sequential, but actions can be defined in any order.
- *unsorted*: The actions are executed in any order.

Typically, each plan can be decomposed into other sub plans again and again until a non decomposable plan is reached (recursive dissolving). Therefore, an *Asbru* plan is a complex hierarchical structure with relationships and interdependencies between single plans. Besides the hierarchical relationships, also temporal dependencies can be defined. To deal with them, each plan can contain time annotations, i. e., attributes that describe the temporal behavior and aspects of a plan. This is realized using four time-points (Earliest Starting Shift (ESS), Latest Starting Shift (LSS), Earliest Finishing Shift (EFS), and Latest Finishing Shift (LFS)) and two durations (Minimum Duration (minDu) and Maximum Duration (maxDu)). Time-points are defined relatively to a specific or abstract reference-point² [Aigner, 2003]. This way of handling temporal uncertainties is equivalent to the *proactive* way of dealing with temporal uncertainties in *Project Management* (see Section 2.1.4).

²As all time-points refer to a reference-point, they are termed as shifts. To have a consistent terminology throughout this thesis, from now on shifts are also termed as time (e. g., Earliest Starting Shift (ESS) means the same as Earliest Starting Time (EST)).

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plan-library SYSTEM "asbru_7_3.dtd">
<plan-library>
  <domain-defs>
    <domain name="controlled_ventilation_domain">
      ...
    </domain>
  </domain-defs>
  <plans>
    <plan-group>
      <plan name="ventilation_plan">
        <intentions> ... </intentions>
        <conditions>
          <complete-condition>
            <constraint-combination type="and">
              <parameter-proposition parameter-name="FiO2">
                <value-description type="less-or-equal">
                  <numerical-constant value="40"/>
                </value-description>
              ...
            </constraint-combination>
          </complete-condition>
          <abort-condition>
            <constraint-combination type="or">
              <parameter-proposition parameter-name="FiO2">
                <value-description type="greater-than">
                  <numerical-constant value="90"/>
                </value-description>
              ...
            </constraint-combination>
          </abort-condition>
        </conditions>
        <plan-body>
          <subplans type="sequentially">
            ...
            <plan-activation>
              <plan-schema name="initial_plan"/>
            </plan-activation>
            <plan-activation>
              <plan-schema name="controlled_ventilation_plan"/>
            </plan-activation>
          </subplans>
        </plan-body>
      </plan>
      ...
      <plan name="controlled_ventilation_plan">
        <plan-body>
          <subplans type="parallel">
            ...
            <plan-activation>
              <plan-schema name="handle_PCO2_plan"/>
            </plan-activation>
            <plan-activation>
              <plan-schema name="handle_tcSaO2_low_plan"/>
            </plan-activation>
            <plan-activation>
              <plan-schema name="handle_tcSaO2_high_plan"/>
            </plan-activation>
          </subplans>
        </plan-body>
      </plan>
      ...
    </plan-group>
  </plans>
</plan-library>

```

Figure 2.18: Example of Asbru 7.3 code: Parts of a clinical treatment plan for artificial ventilation of newborn infants [Aigner and Miksch, 2004].

2.2.4 Graphical Representation of Treatment Plans

As mentioned before, graphical representation is an important concern in *Protocol Based Care*. Developed technical standards need to be communicated to physicians, nurses, and other clinical staff who should work with plans finally.

Graphical representation is an adequate way to communicate a treatment plan in a way needed by clinical staff. If the visualization is done properly, all different characteristic of a treatment plan (like hierarchical decomposition, sequences, or conditions) can be shown within a single visualization application. Such an application allows to represent a guideline in different levels of detail, therefore, on the one hand, a good overview of the whole treatment plan is given, on the other hand, detailed views are possible.

Several common charting techniques to visualize logical sequences and hierarchical data are available and also thinkable (or already in use) in *Protocol Based Care* projects [Tu et al., 2002, Aigner and Miksch, 2004]:

- *Flowcharts*: Flowcharts are a powerful instrument to represent a condition based plan (see Figure 2.19).
- *Structograms, PERT charts and Petri nets*: Techniques to view logical and temporal sequences of a plan.
- *Treemaps*: A common technique to represent hierarchical data.

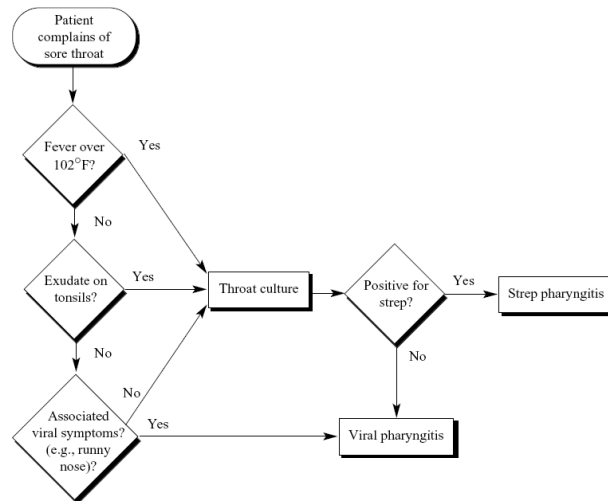


Figure 2.19: Example an flowchart algorithm (detection of a disease) [Hadorn, 1995].

Especially flowcharts are popular in *Protocol Based Care* because most physicians are familiar with this visualization technique, therefore acceptance is high. Each action is walked through depending on conditions placed between them. Flowcharts represent the whole plan, which allows to trace all possible states of a plan.

But, as mentioned before, a clinical guideline is a very complex plan and all of these visualization techniques are not appropriate to join all requirements of treatment plans. Therefore, combinations and extensions of these general techniques are needed to produce a highly sophisticated and practical visualization of treatment plans.

Graphical Representation Tool of Asbru Plans - CareVis

Besides the general demands of a visualization (like performance, intuitive interaction, application safety, etc.), the main challenges in representation of clinical guidelines written in *Asbru* are [Aigner and Miksch, 2004]:

- logical sequences;
- hierarchical decomposition;
- execution order; and
- conditions.

CareVis is a visualization tool for *Asbru* plans, designed to communicate plans to clinicians. It was developed with the aim to solve the mentioned problems within a single application. To meet the demands of physicians a user study was performed (for more details see [Aigner, 2003]), the concepts of *CareVis* are a result of this user study and an evaluation with experts.

Generally, *CareVis* (see Figure 2.21) consists of two different views, a logical view on the left side to represent relationships and hierarchical decomposition between plans, and a temporal view on the right side of the application to display temporal dimensions of plans and patient data upon time [Aigner and Miksch, 2006].

The logical view [Aigner and Miksch, 2004] represents the plan body of an *Asbru* plan. That means, this view is responsible to illustrate the logical order of plans in combination with conditions and the execution sequence (see Figure 2.20). This is reached by applying modified flowcharts, containing some additional information, i. e., literature links or parameter information.

Each plan that contains sub plans can be decomposed within this view. Instead of a rectangular plan symbol, another (logical) view appears within the original view. Besides the logical relations, this allows to show the hierarchical decomposition of a plan [Aigner and Miksch, 2006].

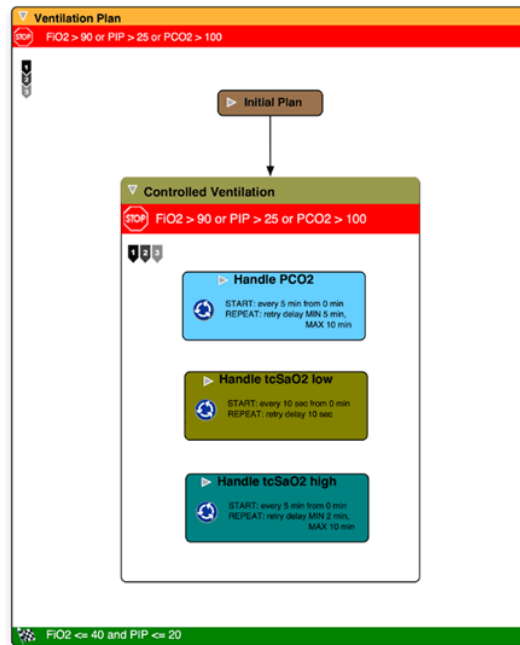


Figure 2.20: Logical view of the given Asbru plan (see Figure 2.18) [Aigner and Miksch, 2004].

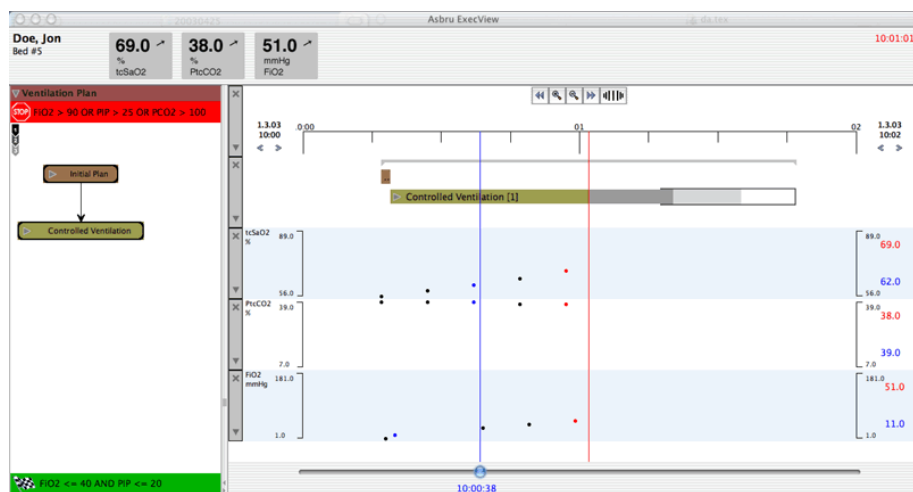


Figure 2.21: Screenshot of the CarVis application, including some patient data [Aigner and Miksch, 2006].

As an *Asbru* plan contains temporal dimensions too, these have to be displayed separately. The temporal view is responsible to visualize these temporal aspects of a selected plan in combination with additional patient data over a period of time. A timescale is shown on the top of the temporal view. All elements below refer to this scale. To visualize a plan regarding to its temporal characteristics the visualization technique *PlanningLines* is applied (see Section 2.1.3 and 3). The temporal view is coupled with the logical view, that means, a selected plan in the logical view is shown in the temporal view too. Besides the temporal aspects of a plan, the hierarchical decomposition is also determinable (see Figure 2.21).

Applying two different views meets all specific demands to visualize *Asbru* plans. Of course this visualization is specialized to *Asbru* plans, but also other *Protocol Based Care* projects can advantage from this approach.

Graphical Authoring Tool for Asbru Plans - AsbruView

AsbruView is a visualization tool and user interface for *Asbru* plans, but it is not specially designed to communicate plans to physicians. The main purpose of *AsbruView* is authoring plans by scientists and medical domain experts, that means creating of new plans and changing of already existing ones [Kosara and Miksch, 2001].

AsbruView consists of three different views, a topological view, a temporal view, and a third view called SOPOView. The topological view (see Figure 2.22) exists of three dimensions: A dimension to represent logical dependencies between plans referring to time, a dimension to display parallel plans (width), and a dimension to display different levels of plans (hierarchical decomposition). Conditions and states of a plan are represented by metaphors of daily life (i. e. , traffic signs).

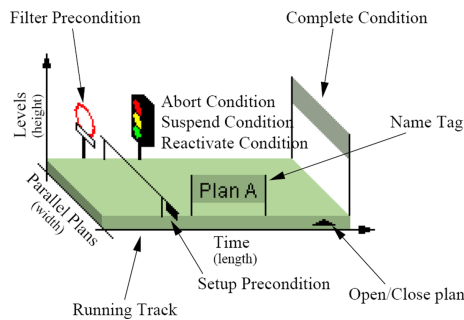


Figure 2.22: Anatomy of the topological view in AsbruView [Kosara and Miksch, 2001].

The temporal view arranges plans along their chronological order. A timescale referring to a given absolute reference-point, is placed on the top of the view. Plans are displayed with a special sort of glyph (see Figure 2.23),

similar to *PlanningLines*. Such a glyph consists of two rectangular bars (standing for Minimum Duration (minDu) and Maximum Duration (maxDu)), and four points (Earliest Starting Time (EST), Latest Starting Time (LST), Earliest Finishing Time (EFT), and Latest Finishing Time (LFT)). Additionally, there exist several variations of this glyph (e.g., different symbols) depending on the amount of available points or durations. All points are defined relative to the reference-point of the timescale.

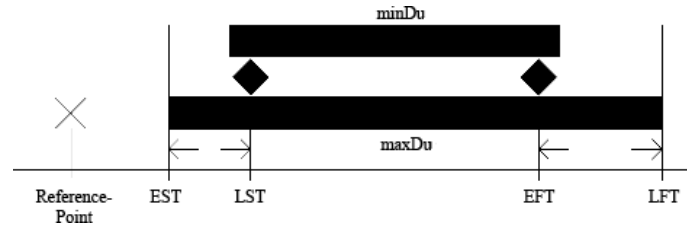


Figure 2.23: Glyph of an plan in AsbruView (adapted from [Kosara and Miksch, 2001]).

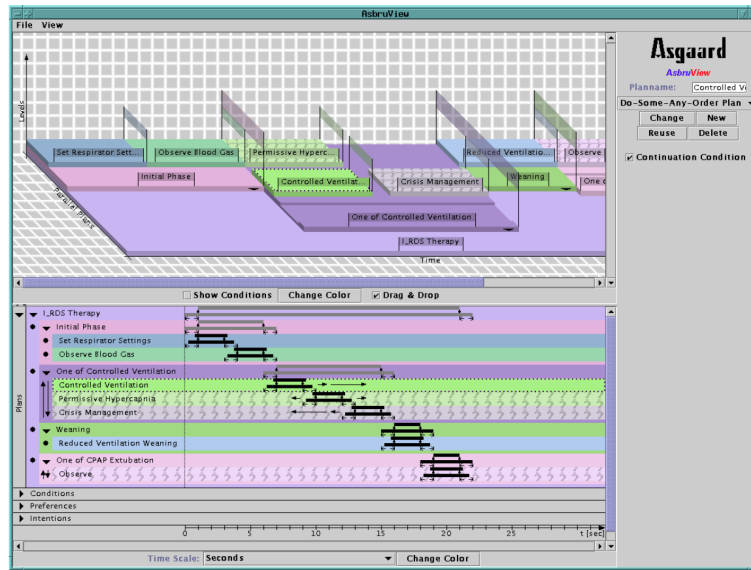


Figure 2.24: Screenshot of AsbruView [Kosara and Miksch, 2001].

The third view is called the SOPOView, which is based on Sets of Possible Occurrences (SOPOs) [Rit, 1986], a method to visualize tasks regarding their temporal attributes. Similar to *PlanningLines* or the glyphs of the temporal view in *AsbruView*, SOPOs also use a set of time-points (EST, LST, EFT, and LFT) and two durations (minDu and maxDu). These points are assigned on two time axes, one stands for starting-times, the other one for

finishing-times (see Figure 2.25). Therefore, any point in this diagram represents a time interval with start and end-time, all given intervals within the gray area are possible intervals of a certain task.

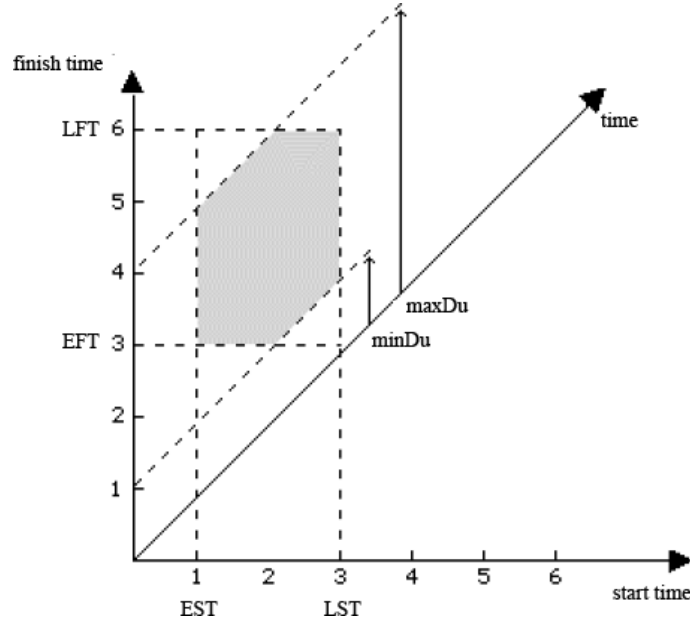


Figure 2.25: SOPO diagram (adapted from [Messner, 2000]).

The SOPOView can be chosen instead of another view to get an additional view of a plan (see Figure 2.26). Even if it is possible to read out all relevant temporal information of a plan, this view has two disadvantages: It is quite hard to depict hierarchical decompositions and parallel plans in a usable way, and this kind of diagram is hard to understand (as a usability study with physician showed) [Kosara et al., 2001].

2.2.5 Related Projects

There are several projects researching different approaches to support *Protocol Based Care*. Below, just a few projects are introduced³.

PRODIGY Project

The PRODIGY Project⁴ was funded by the Department of Health in England. It is a computer-based decision support system, including a guideline

³For more information about Protocol Based Care projects see <http://www.openclinical.org> (accessed on May 30th, 2006).

⁴For more information about PRODIGY Project see <http://www.prodigy.nhs.uk> (accessed on May 30th, 2006).

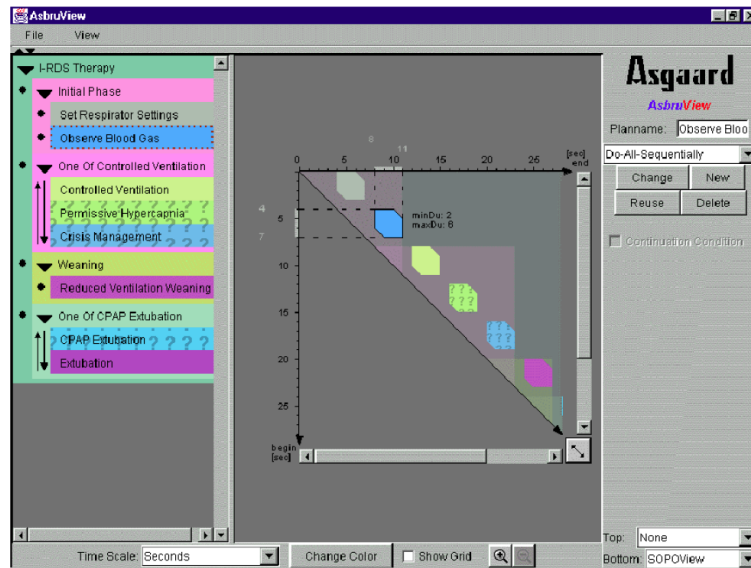


Figure 2.26: Screenshot of the SOPOView in AsbruView [Messner, 2000].

model that is integrated to commercial medical information systems.

PRODIGY is scenario based, that means, depending on patient's condition and actual treating, the model calculates possible scenarios and actions for further treating based on defined rules (conditions). Considering already applied scenarios, the therapy of a certain patient can be seen over time, helping physicians to make decisions [OpenClinical, 2006].

EON

EON⁵, developed by Stanford Medical Informatics, is a suite of models and software components supporting the creation of clinical guideline based applications [OpenClinical, 2006].

Encoding of EON guidelines is done in the Protégé-2000 knowledge-engineering environment (developed by Stanford University too), extended by special demands of healthcare. Protégé is a knowledge framework using plans which are based on ontologies. Graphical editors for creating and editing plans are available. Three different models were developed to meet the requirements of *Protocol Based Care*: A model to represent patient data, an information model containing domain specific attributes and relations, and a model defining the structure of clinical guidelines called Dharma. Further, some applications like a guideline-execution-engine or an explanation-server are available [Tu and Musen, 2001].

⁵For more information about EON see <http://smi-web.stanford.edu/projects/eon> (accessed on May 30th, 2006).

HGML

HGML⁶ - Hypertext Guideline Markup Language, developed by University of Medicine and Dentistry of New Jersey, is a specification of condition and recommendation-elements used in clinical guidelines. Tags are defined in XML/XHTML which can be used in text guidelines.

Several applications were developed to work with guidelines supported by HGML. That includes a web editor for reading guidelines and a patient information system to collect data of patients and receive recommendations for further treatments [OpenClinical, 2006].

As mentioned, one important issue in *Protocol Based Care* is communicating clinical plans to users. This is reached by creating visualizations users can work with. Therefore, the next Section refers to *Information Visualization*, a scientific discipline specialized on the demands of graphical representations.

2.3 Information Visualization

2.3.1 Overview

The visual sense is the most sophisticated ability of humans - making data visible to gain knowledge uses this strength of humans directly [Meyer, 1997]. *Information Visualization* is the scientific discipline to display abstract data in a useful, clear, and comprehensible way.

There are several aims of *Information Visualization*. On the one hand, it is used to provide insight in certain data and support exploration of these, on the other hand, *Information Visualization* is applied to ensure understanding of information that is too abstract, too complex, or contains relationships and dependencies to other information. Generally, the aim of *Information Visualization* is communicating new information and facilitating new insights of the user.

As information can occur in countless variations, there are no general rules how data has to be communicated, it always depends on the domain, *Information Visualization* is applied for. But typically, *Information Visualization* methods follow some common and appropriate paradigms. Such paradigms are available for providing adequate visual representations for the given information, but also for programming techniques to ensure well engineered applications (regarding data storage, data transformation, visual mappings, etc.).

Before mentioning these paradigms, some definitions should be given beforehand to outline the scope of *Information Visualization*.

⁶For more information about HGML see <http://infolab.umdnj.edu> (accessed on May 30th, 2006).

2.3.2 Definitions

Data

The terms information and data are often used interchangeable which leads to confusion. However, these terms are not synonym. In general, data is just a representation of something, often used for computer processing. Data itself has no meaning, meaning primal occurs when data gets interpreted [Joint Publications, 2001]:

“Representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or by automatical means. Any representations such as characters or analog quantities to which meaning is or might be assigned.”

Data can be differed into physical data and abstract data. Physical data refers to the real world (inherently referring to space), for example coordinates or measurable data. Abstract data only exists in a virtual reality, that means it is only valid within a defined space (see Section 2.3.2) depending on the given context [Voigt, 2002]:

“... data that has no inherent mapping to space. Examples for abstract data are the results of a survey or a database of the staff of a company containing names, addresses, salary and other attributes.”

Information

Information is data with an associated meaning [Joint Publications, 2001]:

“... The meaning that a human assigns to data by means of the known conventions used in their representation.”

Information is data that always has to be seen in relation to its context or topic. For example: A single number does not have any importance without an additional context, therefore it can be specified as abstract data. If it is known that this number is a phonenumber, the given number becomes information. But if the same number is declared as a creditcard number, completely different information is extracted out of the same data. How data has to be interpreted is defined via the information space.

Information Space

The context (or topic) of an information is often called information space. This space describes the virtual space a piece of information belongs to.

Such a space can be represented as ranges of valid values, logic mechanism, formal descriptions, etc.

To understand the information space it is important to see the relationships between objects in a semantic context (which can be multidimensional). This often makes it difficult to create an information space and to transmit this virtual space to users in a proper way.

2.3.3 Responsibilities of Information Visualization

Information Visualization is a method to visualize abstract data, information, and knowledge to enable the viewer to see, to browse, and to understand the information. *Information Visualization* is a very complex discipline, covering many other disciplines like informatics, computer graphics, or Human Computer Interaction (HCI). As mentioned in Section 2.3.1, *Information Visualization* is based on visual impressions (mappings) which can be perceived by humans. A well established definition of *Information Visualization* is [Card et al., 1999]:

“... the use of computer-supported, interactive, visual representations of abstract data to amplify cognition.”

Furthermore, six ways are proposed on how visualization can amplify cognition [Card et al., 1999]:

1. *“by increasing the memory and processing resources available to the users,”*
2. *“by reducing the search for information,”*
3. *“by using visual representations to enhance the detection of patterns,”*
4. *“by enabling perceptual inference operations,”*
5. *“by using perceptual attention mechanisms for monitoring, and”*
6. *“by encoding information in a manipulable medium.”*

Therefore, one of the general problems in developing adequate *Information Visualization* methods is to create meaningful visual structures, containing all necessary information a user needs to work with. But, besides representing single information elements, also the information space has to be communicated to users. That means, all relevant dependencies and relations, but also assumptions, margins, validation criteria, etc. must be considered. Concluding, there are always several possibilities on how information can be represented, but only a few are really useful depending on the purpose of a certain visualization.

This is the reason, why *Information Visualization* should be seen in contrast to scientific visualization that i predominantly works with physical data [Voigt, 2002]:

“Information visualization should be seen in contrast to scientific visualization, which deals with physically-based data. This kind of data is defined in reference to space coordinates, which makes it relatively easy to visualize in an intuitive way. The space coordinates in the dataset are mapped to screen coordinates. Examples are geographic data and computer tomography data of a body.”

As in scientific visualization visual structures are clear (given by nature), in *Information Visualization* the design of visual structures has to be explored beforehand.

Exploring the visual appearance of information not only depends on the kind of data. It is very important to consider intended users as well as the tasks the *Information Visualization* should support. That means, the target group of users must be defined clearly, as different groups may demand different representations of the same data. It heavily depends on users’ background, domain knowledge, experience, etc. Further, also the tasks of the *Information Visualization* must be declared beforehand.

But besides visual representations, also interaction techniques and distortion techniques have to be considered which are influenced by target groups and performed tasks again.

2.3.4 Common Techniques

The main goal of *Information Visualization* is to represent information to gain further knowledge. That means, by getting information represented by a visualization, users are able to extract new information or knowledge out of the existing. *Information Visualization* helps to gain knowledge because it enables insight and exploration of certain data.

The task of developing an *Information Visualization* method is also a process of gaining knowledge by exploration and analyzing data, called Knowledge Crystallization.

Knowledge Crystallization

Knowledge Crystallization is the process of collecting information, finding relations and schemata these information fit in, explore which information is really relevant and which not. Finding the optimal schema can be a complex task, but should result a meaningful *Information Visualization* (see Figure 2.27) [Card et al., 1999]:

“The goal of a knowledge crystallization process is to get the most compact description possible for a set of data relative to some task.”

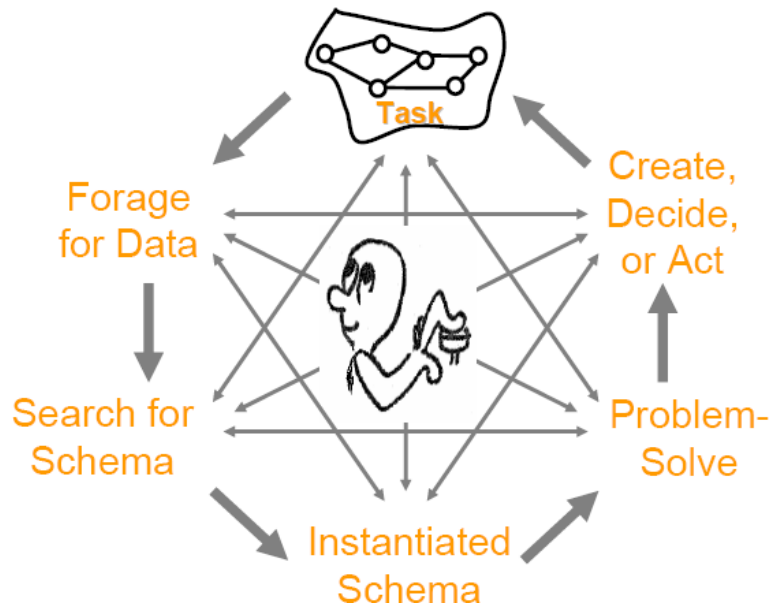


Figure 2.27: Knowledge Crystallization [Card et al., 1999].

Concluding, developing a *Information Visualization* also consists of making a decision how information is presented finally.

Exploratory Techniques

Typically, the more complex information and their space are, the more complex the visualization gets. An *Information Visualization* can apply several techniques to communicate information to users (see Figure 2.28):

- *Visualization Techniques*: Specify how information is represented visually. Common techniques are graph views and hierarchical views, other more complex techniques are icon-based or geometric.
- *Distortion Techniques*: Allow users to see information they are interested in, in a higher level of detail (focus), surrounding information in a lower level of detail (context). Examples for distortion techniques are fisheye-views or hyperbolic trees.
- *Interaction Techniques*: Allow users to control the visualization, like zooming or panning (for more details see Section 2.3.5).

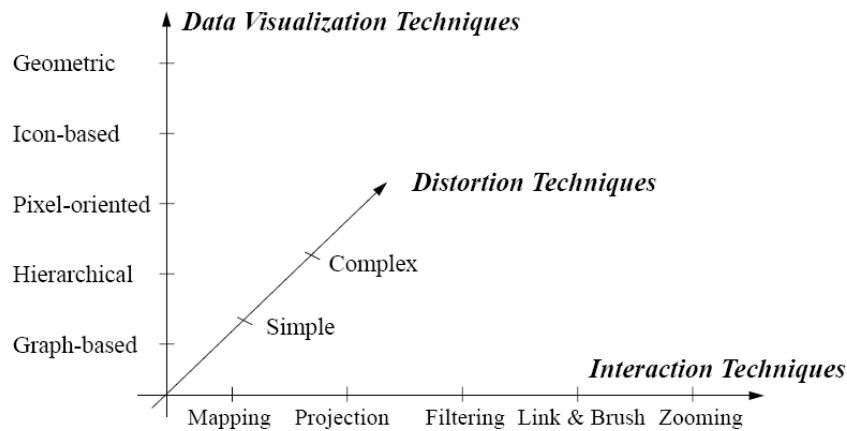


Figure 2.28: Dimensions of Visualizations [Keim and Kriegel, 1996].

Once adequate visual mappings and techniques to represent information are found, the actual technical development starts. There are several methodologies on how an *Information Visualization* can be implemented, like the *Information Visualization Reference Model* [Chi, 2000]. Another common way for developing is the *InfoVis Pipeline*.

InfoVis Pipeline

The *InfoVis Pipeline* is a recommendation on how an *Information Visualization* is structured. The pipeline starts with data on the one end and ends with a finished visualization (view). This procedure is structured into four forms of data, three data transformations and three steps of human interaction (see Figure 2.29).

Starting with raw data, this abstract data is transformed into data tables. Each data row contains all the relevant values of a single data record and additional information about relations or dependencies to other records. Once the table is present, needed records are transformed into visual structures. In the end, the created visual structures are displayed in a view.

This process provides several advantages: First, each single step is performed on user interaction. This does not mean, that a user has to confirm each single step, instead the logic of the application can be implemented for example in a way things are done only on demand (e.g., in hierarchical structures, visual mappings are just loaded one level in advance).

Another advantage is that different views can be applied to the created visual structures, or different visual structures can be applied to a single record of the data table. Therefore, different views can be given based on the same abstract data.

Besides other advantages (e.g., data loading on demand, etc.), this

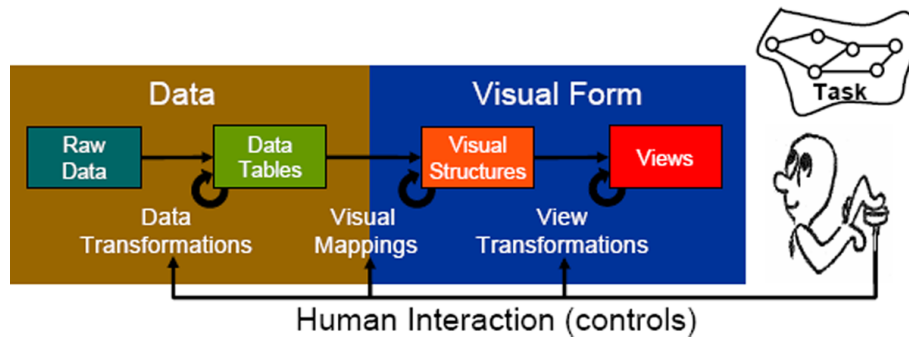


Figure 2.29: InfoVis Pipeline [Card et al., 1999].

methodology generally keeps a visualization expandable for further development.

2.3.5 Interaction Techniques

Interaction is the only way users can control a visualization directly, therefore, developers have to pay special attention to interaction issues. Besides the proper implementation of interaction techniques, it is important to take care of a fast response time whenever the user interacts. Typically, computers allow interaction through keyboard (typing, shortcuts, etc.) and mouse (moves, clicks, gestures), but sometimes also additional system-peripherals are applied.

Panning

Panning is an interaction technique that helps to navigate within a view. Instead of using a bounded view, a user interface that supports panning typically has no bounds. That means, a user can navigate to any direction without limitations given through bounds. Mostly, navigation with panning is realized by dragging the mouse above a view and the view goes along with the movements a user performs.

Zooming

Zooming allows the user to change the size of visual structures on the view. Typically, not only the size of items is affected, but also spaces and distances between items. Technically, each single point of a view is transformed to a new position, depending on the zoom factor. Zooming is relevant, if there are a lot of items or there are items of different size.

Semantic Zooming

Semantic zooming means that not only sizes change when zooming (geometric zooming), but also displayed contents change. When zooming in, additional (more detailed) information is given (focus), when zooming out, too detailed information is hidden (context). This can be applied to written information of single items, but even new visual structures can be added or removed of the view if necessary.

Semantic zooming is used to prevent clutter on the view. Whenever an *Information Visualization* is thought to communicate much information, ways have to be found to avoid an overspill of displayed information. Semantic zooming is a method where users have to decide which information they are interested in are shown in detail as they have interact before all available information is displayed.

2.3.6 Software Toolkits supporting Information Visualization

There are several toolkits supporting *Information Visualization*. Such toolkits provide methods to store data, create visual structures, and implementations of views. Typically, different interaction techniques that may be extended on demand are also provided by visualization toolkits. Besides proprietary toolkits, also open-source toolkits are available.

2.3.7 InfoVis Toolkit

The *InfoVis Toolkit* ⁷ (see also Section 4.2.2) is an interactive graphical open-source toolkit written in Java, developed by the University of Paris-Sud. It is designed to support the creation of advanced 2D visualizations in Java Swing applications. Special attention was paid to process large amounts of data. Its main features are [Fekete, 2004]:

- “*Generic data structures suited to visualization.*”
- “*Specific algorithms to visualize these data structures.*”
- “*Mechanisms and components to perform direct manipulation on the visualizations.*”
- “*Mechanisms and components to select, filter and perform well-known generic information visualization tasks.*”
- “*Components to perform labeling and spatial deformation.*”

Data is stored in a unified underlying data structure based on tables. Tables allow to represent nearly every kind of data structure and improves

⁷For more information about the InfoVis Toolkit see <http://ivtk.sourceforge.net> (accessed on May 30th, 2006).

memory usage and performance. In this point, the *InfoVis Toolkit* is conform with the Visualization Reference Model (see Section 2.3.4).

Some layout and rendering algorithms are predefined, which makes it very easy to create standard visualizations like treemap or graph visualizations. Own, more complex visualizations, can extend existing controls or create new ones. The display can be bound directly to its underlying tables, that means when a value in a table changes, the display is notified about that to be able to react. Also some interaction techniques are already implemented, like zooming and navigation techniques.

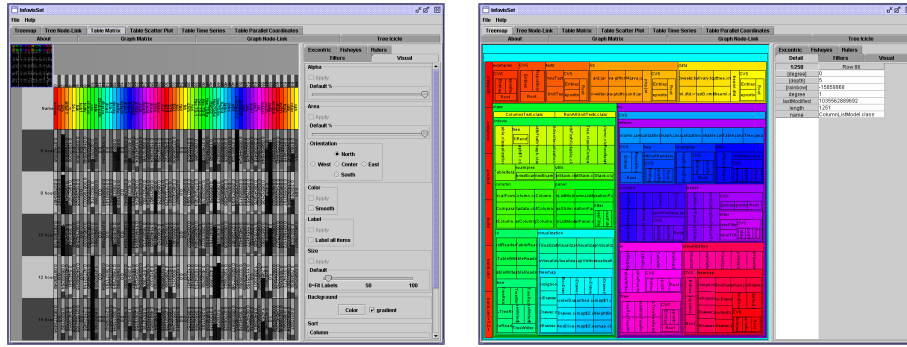


Figure 2.30: A Table Matrix and a Treemap realized with the InfoVis Toolkit [Fekete, 2006a].

2.3.8 Prefuse

*Prefuse*⁸ (see also Section 4.2.3) is an open-source toolkit written in Java, developed by the University of California, Berkeley.

For each abstract data item, a node is created and stored withing a graph structure. Relations between nodes are saved as edges that can be visualized too. All nodes are hold centralized in a container that is able to transform these abstract data into visual structures. Visual structures are created separately out of the abstract nodes. This separation between abstract and visual data is often referred as polythitic design (in contrast to monolithic design where abstract and visual data are concentrated within one object).

Several interaction techniques are already predefined (like zooming or panning). Further, *Prefuse* provides layout and assignment algorithms that place visual structures on the view. Typically, interaction and layout algorithms are defined as action, which are combined within runnable containers

⁸For more information about Prefuse see <http://prefuse.sourceforge.net> (accessed on May 30th, 2006).

that are performed on input events (mouse, keyboard, etc.). Instead of using predefined actions, they can be easily extended to meet own requirements.

Not only abstract and visual data is separated in *Prefuse*, also the painting of items is performed by own objects. Renderers are assigned to visual structures that perform all painting issues of an object. Implemented renderers are held centralized in a factory, therefore a visualization can use different renderers for nodes and edges (i.e., this allows semantic zooming).

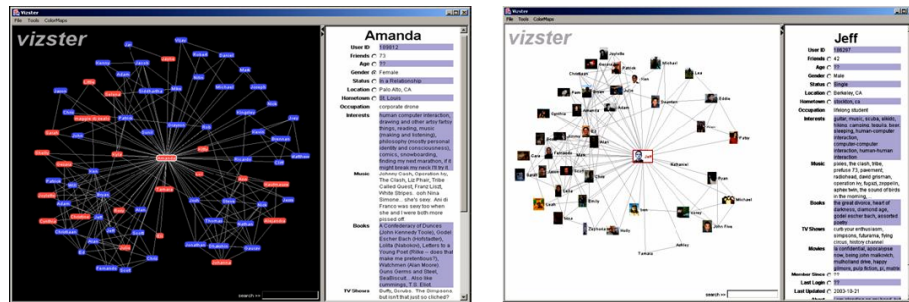


Figure 2.31: Network diagram showing social structures realized with the Prefuse Toolkit [Heer et al., 2005].

2.3.9 Piccolo

Piccolo⁹ is an open-source toolkit written in Java and C#, developed by the University of Maryland. During design of Piccolo, most attention was set to realize Zoomable User Interfaces (ZUIs).

Data is stored in a scene-graph, that means nodes containing data are hierarchically stored together with additional elements like cameras or layers that contains visual transformations or grouping information. A layer contains one or more nodes that can be viewed with a camera. This structure is very complex but provides a lot of possibilities, especially with regards to viewing and navigating.

In contrast to *Prefuse*, Piccolo uses a monolithic design. A node contains abstract as well as visual data, further painting issues are performed by a node itself. This design ensures fast results when developing an *Information Visualization* but extensions of functionality are more difficult to realize.

Piccolo was developed to support developing 2D graphic programs with special demands in zooming. Piccolo provides smooth animated zooming, animated panning, and a lot of other effects. Other functionality provided includes efficient repainting of the screen with bounds management, event handling and dispatching, picking, and layouting.

⁹For more information about Piccolo see <http://www.cs.umd.edu/hcil/piccolo> (accessed on May 30th, 2006).

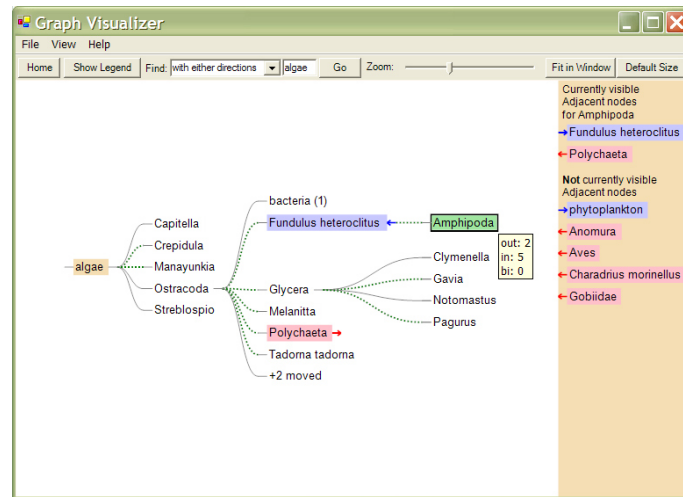


Figure 2.32: Tree realized with the Piccolo Toolkit [Piccolo, 2006].

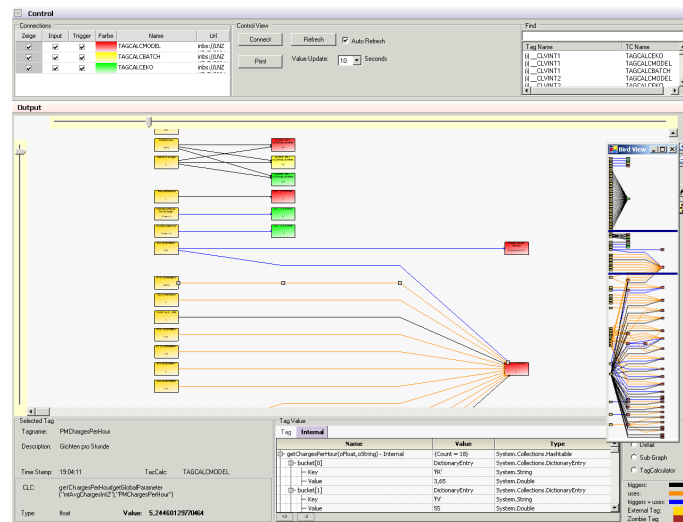


Figure 2.33: Graph realized with Syncfusion Diagram.

2.3.10 Syncfusion Essential Diagram

In general, Syncfusion¹⁰ is not a *Information Visualization* toolkit, but a commercial component library for .NET applications. However, one of its components, the Essential Diagram provides a highly interactive form to represent visual structures. This makes the Essential Diagram to an interesting component doing *Information Visualization* in .NET.

Essential Diagram uses the Model-View-Controller (MVC) design pattern to clearly separate data, presentation, and user interaction. Visual data structures are hold in a model, a view renders the model to the screen, and a controller handles user inputs.

Data is stored hierarchically, that means each visual structure (node) can contain parents and children. Besides the logical structure, also a graphical structure can be applied by using edges (see Figure 2.33).

In practice, Essential Diagram is easy to use for smaller visualizations. Realization of complex visualizations with special demands is a quite hard task, because extension of fundamental classes (like the model or the view) is not or only with additional effort possible.

The major goal of this work is the development of a prototype representing plans with the concept *PlanningLines*. Therefore, after outlining and discussing surrounded scientific disciplines, next Chapter describes this concept and its notation in detail.

¹⁰For more information about Syncfusion see <http://www.syncfusion.com> (accessed on May 30th, 2006).

Chapter 3

PlanningLines

Treating temporal uncertainties is an important task in planning. *Information Visualization* in combination with PERT provides a way to deal with them in a comfortable and practical manner (see Section 2.1.5 and 2.3.3), but temporal facts are only represented textually. Even if PERT is a widely spread and accepted technique, the textual representation may be a disadvantage, especially when mapping large and complex plans. Getting an overview and identifying problems can be hard work. Therefore, PERT is usually used in combination with other charting techniques like Gantt to minimize these disadvantages.

A possible alternative to PERT is the charting technique *PlanningLines*, which represents temporal attributes in a graphical way. Even if the concept *PlanningLines* was originally developed to represent medical treatment plans, it is a suitable and adequate technique for any kind of plan regarding to time.

3.1 Comparison to PERT and Gantt

As *PlanningLines* were developed to treat temporal uncertainties in combination with a clear and meaningful representation of tasks, there are some advantages compared to PERT, which is the most common charting technique used to deal with temporal uncertainties:

- *Indeterminacies*: PERT is a visualization technique that uses textual descriptions of temporal facts. Only tasks and their interdependencies are shown visually (a common variation also illustrates hierarchical decomposition). Missing attributes of a task do not influence the visualization at all, only some empty text-fields allow to identify these indeterminacies. The notation of *PlanningLines* considers missing values by communicating all given (and also missing) temporal conditions of a task to users visually. To treat all possible combinations of tempo-

ral attributes, also special constellations of *PlanningLines* can occur. This allows users to spot eventual problems at a glance.

- *Chronological Interdependencies*: PERT charts only represent logical relations between tasks. Chronological interdependencies must be gathered by textual information. As *PlanningLines* uses a timescale to which all tasks refer, all coherencies within a plan can be determined easily.
- *Perceptibility of Tasks*: All tasks represented with the PERT notation have an identical look. Visually, it does not make a difference how long a task lasts or if an activity contains temporal uncertainties. This representation makes it hard to distinguish tasks by their temporal facts (which is often equivalent to the importance or value of a task). Especially in large and complex plans, identifying possible problems and coherencies or getting an expressive overview may be hard to achieve. *PlanningLines* allows to retrieve all important information at a glance.

All these problems are caused by the underlying notation of PERT charts which basically is based on textual descriptions. Relevant temporal information must be gathered by users themselves from these descriptions. The accumulation process can be hard work, therefore in practice only slack times between tasks are considered mostly. But, it is easier to retrieve detailed temporal information of textual description. Visual information may be easy to conceive but getting details may be difficult.

To map tasks referring to time, Gantt is the leading charting technique used. Gantt charts allow a well structured and clear visual representation of tasks along time. This technique is based on a timescale all task refer to, therefore the start and end of a task can be identified at a glance.

Coherencies and durations of tasks are easy to determine, an extension of the notation even allows to represent hierarchical decomposition. As the visual sense is the most sophisticated ability of humans, Gantt charts are usually conceived quickly and offer a clear and meaningful overview of a plan. Nevertheless, insight is communicated too when using hierarchically decomposed plans. As established this technique is, there is no notation to consider temporal uncertainties in Gantt charts so far.

PlanningLines tries to combine the advantages of PERT and Gantt charts. That means, tasks are represented as usual in a Gantt chart, containing additional temporal attributes as used in PERT that are represented visually.

3.2 Requirements

Originally, the concept of *PlanningLines* was designed to enable a temporal view of treatment plans written in *Asbru* for clinical staff like physicians

or nurses. Both, application area and target group, require some special demands a charting technique must fulfill.

First, as most medical staff is not familiar with complex visualization techniques, an essential demand of *PlanningLines* is a visual form domain experts can work with. This means, the notation of *PlanningLines* must be clear, easy to understand, and relevant information have to be communicated in a way users can conceive them fast.

Further, a medical task within a treatment plan can contain a lot of complex data, including temporal uncertainties. As *Asbru* uses several loose attributes (e.g., possible start and end-times or possible durations a task can last), the notation of *PlanningLines* must support all occurable combinations of these attributes (including missing ones). All temporal attributes should be represented in a clear visual form instead of textual descriptions.

Also, a treatment plan written in *Asbru* usually contains a lot of different single tasks that can be interrelated. Such coherencies can be logically (e.g., Task B follows Task A) or hierarchically (e.g., Task B consists of Task C and Task D). These relationships also have to be communicated to users to ensure a correct understanding of the whole plan.

Concluding, besides the notation also the interaction possibilities of an application that uses *PlanningLines* are of interest. Well considered interaction techniques ensure a useful and meaningful view, on the one hand, and usability and acceptance increases on the other hand.

Even if these requirements are intended for the medical domain, also demands of planning in general are covered therewith. The original concept was expanded and modified by the demands of *Project Management*, so *PlanningLines* are suited to fulfill issues of *Project Management* too.

3.3 Design Concept

Considering the mentioned requirements, two major issues can be identified¹: first, the representation of a single activity or task, second, the representation of multiple tasks belonging together (e.g., parts of a plan or a whole plan) [Aigner et al., 2005a].

As the concept is thought to be applied in a visualization, there is several other issue that may affect the representation (e.g., graphical extensions and applied interaction techniques). But such concepts are not strictly defined, therefore, developers of a visualization may combine the original concept with other concepts and interaction techniques (see Section 4).

Representing a single task is realized using a glyph (*“A symbol, such as a stylized Figure or arrow on a public sign, that imparts information nonver-*

¹All shown images in this Chapter are design concepts. The representation used by the prototype may differ in some details.

bally” [Morris, 2000]). Generally, this glyph provides the same information graphically as an element of PERT does textually.

As all information refer to time, such a glyph always depends on another graphical element representing time. Typically, time is illustrated by using a more or less detailed timescale to which all glyphs of a plan are related.

The second major goal considered in the design concept is the representation of multiple glyphs and their coherencies among each other. There are two kinds of coherencies that can be distinguished, logical relations between tasks or tasks that are composed hierarchically. All developed concepts falling in this domain support a well structured view on a plan, facilitating the identification of critical areas, helping to understand relations, and allowing a comparison of tasks and activities.

3.3.1 PlanningLine Glyph

The glyph used to represent a task of a plan with *PlanningLines* has to cover all temporal attributes of a task. To reach this, the concept of *LifeLines* [Plaisant et al., 1998] was extended by several elements. A *LifeLine* is a horizontal bar showing time-oriented data. The new visual element which is resulted of these extensions is called *PlanningLine*.

The most important extension that was applied is the possibility of representing more time attributes than just the begin and end of a task. A *PlanningLine* glyph consists of several graphical elements nested into each other (see Figure 3.1). These additional elements allow to visualize all attributes used in a typical PERT task.

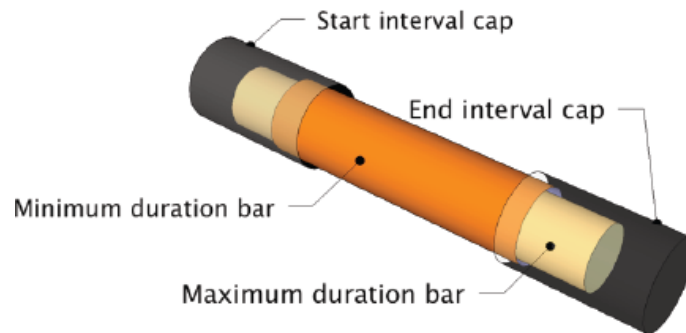


Figure 3.1: 3D concept of a PlanningLine glyph.

Instead of using an explicit start and end-point, intervals are applied to *PlanningLines*. These intervals are represented with caps bounding both start and end of a *PlanningLine*. Encapsulated between these intervals are two bars, illustrating a minimum and maximum duration of a task. Combinations of these elements provide multiple ways of modeling temporal

facts, and, they are also capable to illustrate temporal uncertainties and indeterminacies.

Temporal Attributes

A glyph consists of following temporal attributes [Aigner et al., 2005a] (see Figure 3.2):

- Starting-interval (Earliest Starting Time (EST) and Latest Starting Time (LST));
- Minimum Duration (minDu) and Maximum Duration (maxDu); and
- Ending-interval (Earliest Finishing Time (EFT) and Latest Finishing Time (LFT)).

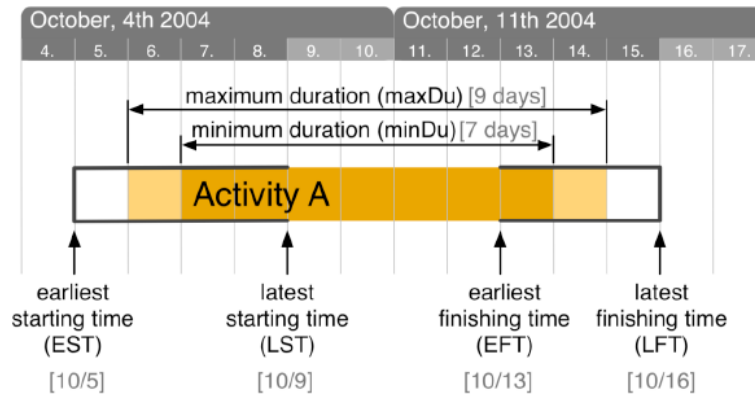


Figure 3.2: Visual representation of a PlanningLine.

These temporal facts imply that the actual starting-time of an activity might be any point between EST and LST, the actual finishing-time between EFT and LFT. Also the actual duration of an activity is not given exactly, it might be any timespan between minDu and maxDu.

The mental model (see Figure 3.3) for the glyph illustrates how the time attributes have to be seen: Both caps representing possible intervals of actual start or end hold the duration bars in between. These caps are fixed (bound to a timescale), as they are given concretely by EST, LST, EFT, and LFT. The duration bars are only defined by spans, they can be moved to left or right as much as the bounding caps allow. Further, the minimum duration bar must be within the maximum duration bar.

Time annotations can either be given absolutely referring to a timescale (e. g., 25 June 2006), or relatively to a reference-point (e. g., 5 minutes after

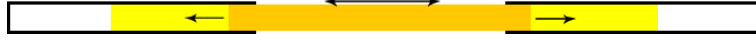


Figure 3.3: Mental model of a PlanningLine.

the end of Activity X). This is especially useful in medical treatment planning, as plans are usually reused (e. g. , composing a plan of some predefined subplans).

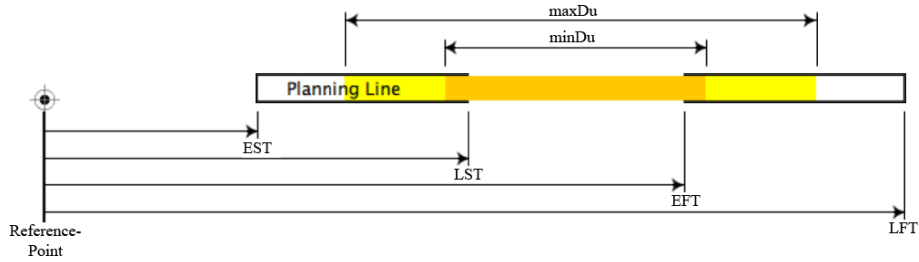


Figure 3.4: Model of a PlanningLine with relative time annotations (adapted from [Aigner, 2003]).

Temporal Attribute Constraints

To obtain a valid set of temporal attributes, some logical constraints must be complied with. These constraints can be easily derived by considering the underlying mental model of a *PlanningLine*. Formally, these constraints are [Aigner et al., 2005b]:

- The interval between the Latest Starting Time (LST) and the Earliest Finishing Time (EFT) defines the smallest possible and the interval between the Earliest Starting Time (EST) and the Latest Finishing Time (LFT) defines the largest possible time window for the duration of an activity.
- For each single time-point in the starting-interval $[EST, LST]$, there must exist at least one duration out of $[\min Du, \max Du]$, which allows the finishing-interval $[EFT, LFT]$ to be reached.
- Each single time-point in the finishing-interval $[EFT, LFT]$ must be reachable by at least one duration out of $[\min Du, \max Du]$ from the starting-interval $[EST, LST]$.
- Each duration must connect one instant in the starting-interval with one instant in the finishing-interval.

Regarding to the mental model, the interval caps hold both duration bars. That means, each duration bar must be at least as large as the spaces between LST and EFT, otherwise the bar would fall out. Furthermore, no possible duration can last longer than the given caps allow as the bars would not fit in. Therefore, following general logical rules can be given to maintain a valid *PlanningLine* [Duftschmid, 1999]:

- $LST \geq EST, LFT \geq EFT, EFT \geq EST, LFT \geq LST, \max Du \geq \min Du$
- $\text{minimum}(EFT - EST, LFT - LST) \geq \min Du \geq EFT - LST$
- $LFT - EST \geq \max Du \geq \text{maximum}(EFT - EST, LFT - LST)$

Indeterminacies and Special Constellations

To obtain a valid *PlanningLine*, the set of time attributes must not be complete. Even if one or more attributes are missing, it is often possible to create a complete *PlanningLine* anyway. Depending on the given logical constraints, missing attributes can be calculated manually if they can be determined from others. Nevertheless, the logical rules only define valid intervals of durations. Therefore, all calculations are based on following assumptions:

1. Maximum Duration ($\max Du$) defines the span between EST and LFT (although the rules would also allow a shorter duration).
2. Minimum Duration ($\min Du$) defines the span between LST and EFT (even though a longer duration would be possible).
3. In case of overlapping intervals, the shortest possible duration between the intervals is taken for the Minimum Duration ($\min Du$).

As calculations are only possible when at least one duration is given, firstly, it must be tried to extract them out of the given time-points if one or both are missing. The $\min Du$ can only be calculated if LST as well as EFT are defined:

$$\min Du = EFT - LST \quad (3.1)$$

In case of overlapping intervals and defined EST or LFT, the second logical rule is assumed for the $\min Du$:

$$\min Du = \text{minimum}(EFT - EST, LFT - LST) \quad (3.2)$$

The $\max Du$ is calculated with the LST and EFT. If one of these is missing, it is not possible to determine a plausible duration:

$$\max Du = LFT - EST \quad (3.3)$$

Once, at least one duration is predefined in the plan or was calculated considering above given rules, other missing time-points are tried to calculate. Typically, outer points of intervals are calculated considering the $\max Du$, inner points with regards to the $\min Du$. Appendix A shows all constellations of a *PlanningLine* where calculations are possible. Further, the applied formulas for each constellation are given too (regarding to the Equation numbers).

In case of LST or EFT is missing, following rules are applied:

$$LST = EFT - \min Du \quad (3.4)$$

$$EFT = LST + \min Du \quad (3.5)$$

If both, LST and EFT, are missing, but, a $\min Du$ and EST or LFT is given, following rules are applied in the given order. That means, if EST is defined, it is assumed that there is no starting-interval. In this case, EFT must be reached in Minimum Duration ($\min Du$). Otherwise, an indeterminate ending-interval is assumed and the LST of the starting-interval is calculated:

$$EFT = EST + \min Du \quad (3.6)$$

$$LST = LFT - \min Du \quad (3.7)$$

In case of EST or LFT is missing, a calculation of $\max Du$ was definitely not possible, therefore, an existing Maximum Duration ($\max Du$) was defined in the plan. Nevertheless, to stay consistent in all calculations this defined duration is also seen as maximal possible duration a task can last. Therefore, following rules are applied:

$$EST = LFT - \max Du \quad (3.8)$$

$$LFT = EST + \max Du \quad (3.9)$$

Such calculated attributes are displayed as if they were present. Also, if either the start-interval or the end-interval is indeterminate (EST or LFT is missing and calculation is not possible), instead of the caps little diamonds illustrate the only given time-point (see Figure 3.5). If neither calculations nor diamonds make it possible to obtain a valid *PlanningLine*, it will not be displayed.

Regarding to the given rules and considering handling of missing attributes, some special constellations of a *PlanningLine* can occur. These constellations reach from overlapping intervals to the use of diamonds instead of caps [Aigner et al., 2005b] (samples with explanations of special constellations, see Figure 3.6).



Figure 3.5: PlanningLine with no starting-interval.



Figure 3.6: Examples of special constellations of PlanningLines: (a) shows overlapping intervals, (b) has no maximum duration, (c) earliest and latest starting-time is the same, (d) missing latest finishing-time in combination with a not defined maximum duration.

3.3.2 PlanningLines Display

The responsibility of the display is the representation of multiple activities. Typically, a plan consists of several interrelated tasks. Once all tasks are fulfilled successfully, also the plan itself has reached its end. Therefore, when visualizing a plan, it is important to take care of a meaningful and clear representation of the single tasks themselves, but the representation of their coherencies among each other must be considered too.

Besides dealing with logical relations, the concept of *PlanningLines* also contains the handling of hierarchically decomposed activities. As practice has shown, it is useful to split bigger activities or tasks into smaller pieces of work (called subtasks). On the one hand, this method allows to view a plan in different levels of detail, on the other hand, smaller tasks are better to manage in reality.

As *PlanningLines* were originally designed to represent *Asbru* plans which are composed hierarchically, a notation of representing hierarchies was applied to *PlanningLines* from the beginning. But also in *Project Management* hierarchical decomposition is a common technique to map complex project plans.

Arrangement of PlanningLines

Generally, the representation of *PlanningLines* uses a two-dimensional chart where the horizontal axis refers to time and along the vertical axis *PlanningLines* are arranged (in case of hierarchical decomposition also the depth of tasks). This view is exactly the same view as used by hierarchical Gantt charts.

Time is represented with a timescale. This scale is placed in the top of the view and all *PlanningLines* relate to this timescale. The entire view depends on the actual interval represented by the scale. That means, the horizontal alignment (position and dimension) of every *PlanningLine* is designated by the actual state of the scale.

The single *PlanningLines* are listened along the vertical axis, starting with the root activity on the top. The arrangement depends on the used layout algorithm, but typically items are sorted downwards along their actual start. However, as the concept *PlanningLines* also supposes logical relations and hierarchical decompositions (see Section 3.3.2 and 3.3.2), some restrictions in alignment are given as these coherencies force a grouping of related *PlanningLines*.

Timescale

The timescale on the top of the view is the central element of the view [Aigner, 2003]. In fact, a scale consists of two points, a start-point on the left of the view and an end-point on the right. The interval represented by these points can be in the past, the present, or in the future. The interval is separated by several vertical lines with additional captions that represent points in time. The distance between these ticks depends on the granularity of the scale (see Figure 3.7). Granularity is the separation of a time interval into logical units. The timescale shown in Figure 3.7 has a granularity of minutes for the bigger ticks and a granularity of 15 seconds for the smaller ones.

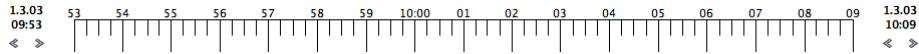


Figure 3.7: Sample timescale with the granularity minutes (adapted from [Aigner, 2003]).

A timescale in an *Information Visualization* typically is dynamic. That means, the user can affect which interval is actually represented. Going conform with the scale, the entire view of a plan has to be adapted whenever a user interacts and vice versa. Only *PlanningLines* containing points which fall into the interval the timescale actually represents are visible. Also the actual width of a *PlanningLine* may change on users' interaction as it also depends on the actual interval.

As the timescale can map every possible time interval, also granularity has to be dynamic. Granularity can be any possible and adequate time span (e.g., milliseconds, hours, days, years, 50 years, etc.). It depends on the application if a user can choose granularity itself or if it is set automatically (best fitting to the interval actually represented).

Logical Relations

Due to possible logical relations of tasks among each other, the display has to care about a representation of these coherencies. In contrast, also loose activities can exist, that means their execution start is not dependent on any other activities. Two different kinds of logical relations are supported by *PlanningLines*.

- *End/Start*: The actual start of an activity is only possible when one or more other activities have already finished. Such a logical relation is necessary when the actual activity depends on the output or resources of its predecessors. If an activity has more than one predecessor, completion of all of them is assumption before the execution of the activity can start.
- *Start/Start*: The actual start of an activity triggers other activities to start. This can be useful when two or more activities share the same resources.

PlanningLines uses connecting arrows to display logical relations (see Figure 3.8). An arrow starts from the predecessor and points to the successor, it also indicates the direction of the relation. It is very important to take care of a clear illustration of these coherencies, because logical relations can easily cause problems in fulfilling a plan. By using arrows, users of the visualization can spot these areas at a glance.

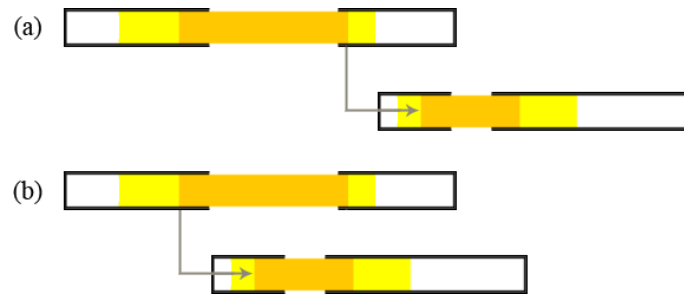


Figure 3.8: Sample of a (a) Start/End relation and a (b) Start/Start relation.

Hierarchical Decomposition

Hierarchical decomposition is given whenever a task is split into smaller pieces of work. That means, the activity stands for a summary of other tasks. The depth of the resulting hierarchy is not restricted in any way, so it is ensured that large and complex plans can be visualized too.



Figure 3.9: Hierarchical decomposition displayed with *PlanningLines*. (a) shows a collapsed PlanningLine noticeable by the expanding-symbol, (b) shows an expanded PlanningLine as summary bar.

A *PlanningLine* containing other activities is marked with a small triangle on its left side. This symbol is well known from different applications representing expanded or collapsed states of leafs in a tree view. Once a *PlanningLine* is expanded, a summary bar appears instead of the *PlanningLine*. A summary bar (see Figure 3.9) is a line with triangular ends. As this bar refers to the timescale too (its width is the same as the task lasts), it is possible to recognize its containing subtasks by following the vertexes of the ends.

This representation also allows to illustrate deeper hierarchies as further summary bars have to fit in their particular parents' summary bar.

Expanding or collapsing such a hierarchy is performed by users. Therefore, users can choose the level of detail on how a plan is shown. An expanded task provides more details than a collapsed tasks. Therefore, users can choose the level of detail of certain areas of the whole plan.

3.4 Example of a Project Plan

Figure 3.10 shows a part of a project plan of a construction work using *PlanningLines*. Most of the described notation is used within this plan. The timescales' granularities are weeks and days, weekends are highlighted additionally.

This plan uses hierarchical decomposition to combine activities. In case of collapsing the whole plan, only two activities would remain: the activity "Carcass" (including all work from "Earthworks" to "Roof"), and the following work "Screed". Withing the first summary bar, there is the composed activity "Fundament/Walls" containing "Foundation" and "Walls and Ceilings".

All activities are connected with arrows which describe logical relations among each other ("End/Start" relations). Especially the task "Screed" is interesting in this context: it follows "Windows/Doors" as well as "Roof". That means that the whole "Carcass" must be complete before the actual start of "Screed".

Different forms of temporal uncertainties are also included. When comparing the two activities "Windows/Doors" and "Roof" one can see at a

glance that the activity “Windows/Doors” is afflicted with more uncertainties regarding begin and end than activity “Roof”. On the one hand, “Windows/Doors” has a lot of slack time and large beginning and ending-intervals whereas activity “Roof” has a fixed beginning and much less uncertainty in its finishing-time. On the other hand, activity “Roof” is more indeterminate in terms of its duration compared to “Windows/Doors”.

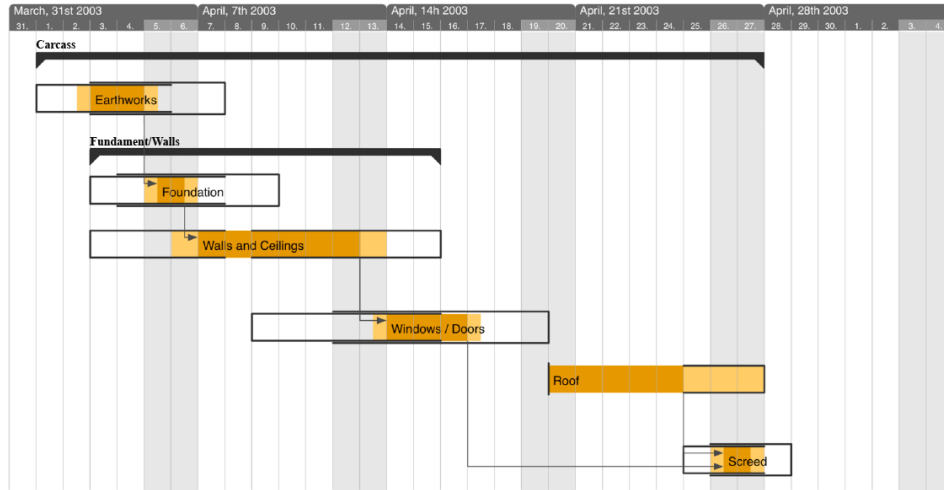


Figure 3.10: Example of a simple project plan, including logical relations and hierarchical decomposition (adapted from [Aigner et al., 2005a]).

3.5 Temporal Uncertainties

The notation of *PlanningLines* allows an exact treating of temporal uncertainties. It enables the creation of a plan with a flexible time schedule. Applied time annotations in combination with logical relations and hierarchical decomposition allow the representation of nearly every thinkable plan that refers to time.

Generally, time is an important factor in planning. Thinking about temporal uncertainties does not guarantee a perfect execution of a plan. But, that's for sure, thinking about temporal facts beforehand will cause better elaborated plans than ignoring them. The use of flexible start and end-points in the concept *PlanningLines* supports creation of flexible plans. A graphical representation helps to grasp sequences and relations. The resulting perceptions of the graphical representation flows back into planning itself, but they are also useful while execution of a plan to track progress.

PlanningLines use a *proactive* way of treating temporal uncertainties, that means possible uncertainties are considered before the actual execution of a plan starts (see Section 2.1.4). This way demands some flexibility

applied to the whole plan (therefore to most of the single activity too).

PlanningLines provide the needed flexibility by using a complex set of time annotations. These attributes allow defining floating begins and ends of activities. That means, instead of using a fixed begin and end an interval indicates a time span when an activity can start or end. This flexibility is used through the whole plan (e.g., if one activity has a floating end, also its successor may consider this circumstance using a floating start).

Besides these intervals, also the given durations used in *PlanningLines* help to apply flexibility to a plan. Typically, they are used to define the finishing-interval once a start-interval is given. But in more sophisticated plans, these durations also can be independent from the intervals (as long as they fulfill the logical constraints (see Section 3.3.1)). In this case, these durations help to estimate the probable end once an activity has started, or even can help to set priorities of tasks (e.g., activity has started very late, to keep time schedule of the whole it is necessary to complete this task in minimum duration).

Typically, planning does not consider temporal uncertainties. In practice, often fixed durations are assigned to tasks. But, estimations of durations are mostly ascertained using an interval (e.g., “this work lasts between three and five days”). When no flexibility is applied to a plan, any duration of the estimated interval is assumed. If perfect conditions and circumstances are expected, the smallest duration flows into the plan, otherwise a longer duration.

PlanningLines allows to consider each point within an ascertained interval. If more temporal information are available (e.g., estimation of durations and time-slot for execution), these facts are also taken into account. But it is also possible to work with fixed time slots and fixed durations as used in Gantt charts for example. This can be obtained by equalizing related attributes (EST and LST, EFT and LFT, minDu and maxDu).

Considering temporal uncertainties with *PlanningLines* provides a visual impression of all activities including all possible characteristics of them. As all attributes are represented graphically in a logical order, characteristics of tasks can be seen at glance. But the visualization can also put only parts of a plan in perspective. Therefore, possible problems not seen at design time can be spotted easily, so solutions for these problems can be worked out before execution starts.

3.6 Discussion

The concept *PlanningLines* is an alternative to common charting techniques. It combines the powerful view of Gantt charts with the exact presentation of temporal uncertainties used in PERT. As all information are provided graphically without losing details, this technique is suitable for use in professional

Project Management too.

Comparing PERT with *PlanningLines*, a user evaluation showed that users are capable to use and apply *PlanningLines* easily [Aigner et al., 2005b]. On the one hand, detailed questions about temporal uncertainties could be answered, even if PERT is advanced in this point. On the other hand, critical sections are spotted easier using *PlanningLines* than PERT. This result was predictable, as it is harder to extract exact information from graphical content than textual content. However, this problem is solvable using an advanced user interface providing additional textual information. Concluding the evaluation study, *PlanningLines* are easy to understand and are accepted by the participants as a practical technique.

After presenting *PlanningLines* and its notation in detail, the next Chapter describes the requirements of the prototype. There, also applied techniques of *Information Visualization* are illustrated. These techniques help to navigate within a plan, but also provide additional textual information for a better representation of temporal attributes.

Chapter 4

Prototype Design

After the theoretical part and a detailed Section of treating temporal uncertainties with *PlanningLines*, the practical part of this thesis, a prototype applying *PlanningLines* to *MS-Project* and *Asbru* plans, is now outlined. The implementation of the prototype was done within the scope of the *Asgaard Project* following the intention of resulting a sophisticated and usable application. There are several purposes this application and its development should meet:

- Demonstrating the power of *PlanningLines* within a practical *Information Visualization*.
- Extending the notation of *PlanningLines* with an adequate user interface and interaction techniques, helping to communicate all information to users.
- Providing a base for the temporal view of *CareVis* (see Section 2.2.4).
- Evaluating and testing of open-source toolkits for graphical visualizations.

Generally, the toolkit was developed using Java¹ SDK 1.4 in combination with the development platform Eclipse².

4.1 Requirements and Environment

Meeting the requirements of the prototype can be divided into several different areas. Besides the given development environment of the *Asgaard Project*, used toolkits for managing data and graphical representation have

¹For more information about Java see <http://java.sun.com> (accessed on May 30th, 2006).

²For more information about the Eclipse development platform see <http://www.eclipse.org> (accessed on May 30th, 2006).

to be cleared up beforehand. Choosing suitable toolkits depends on factors like the selected data structure of storing tasks and their information, planned interaction techniques, performance issues, and the possibility of manipulating a given toolkit in a way needed by the application (extensions, compatibility, etc.).

4.1.1 Basic Environment

The application must be platform independent, it should be executable using Microsoft, Apple, and Linux operating systems. Thinking about its proper purpose, a graphical visualization, the need of dealing with complex data, the surrounding environment of the *Asgaard Project*, and the availability of open-source toolkits Java seems to be the adequate programming language.

At the time of development start, Java 1.4 was in use for writing applications for the *Asgaard Project*. To stay compatible, this version was also used for this prototype even though newer versions were already available.

Besides standard Java and Java Swing components, also open-source components are accredited to ensure fast development on the one hand and to get insight views of them for further development on the other hand.

In the end, there should be an executable Java Archive (Jar) running on the mentioned operation systems without further requirements than an installed Java-engine.

4.1.2 General Requirements

As in every graphical visualization, special attention has to be paid towards performance. Often executed algorithms have to be optimized, painting of components may not waste too much time, and especially repainting should be executed only when necessary.

Another issue is memory usage and management of data structures. Complex data structures should be created only when needed, furthermore, fast access must be ensured to often used data. A large amount of memory can also affect performance, and often memory usage can be minimized when an application is well developed. Therefore, abstract data and visual structures are only created when needed. As the plans are hierarchically decomposed, children of a task are only loaded on demand, that means, a user must explicitly request to see respective subtasks.

As this application represents a prototype with potential to be once part of a sophisticated practice tool supporting *Protocol Based Care* (see *Care Vis*, Section 2.2.4), it is important to choose an architecture that allows further development easily. Therefore, interfaces must be defined well, classes must be separated strictly, and the classes must be written in a way they can be extended easily. The demand of flexibility and ability of extension also requires well defined namespaces to ensure a logical structure of objects.

Information Visualization Pipeline

The development process should follow the Information Visualization Reference Model (also called *InfoVis Pipeline*, Section 2.3.4). On the one hand, this model defines internal operations of an *Information Visualization* like data transformations, and on the other hand, it sets clear points where a user can interact.

The recommendations of the *InfoVis Pipeline* affect the whole architecture of an application. Therefore, the consideration of this model from the beginning on is essential.

To go conform with the *InfoVis Pipeline*, using of toolkits must be evaluated toward their possibilities to stay within the model, as the storage and transformation of data is strictly regulated by it.

Data Source

The prototype should work with two different data sources. On the one hand, it should be able to read and represent *MS-Project* plans, on the other hand, it should also work with *Asbru* plans.

Reading *MS-Project* plans requires a Java library for translating a *MS-Project* file. The open-source library MPXJ³, developed by Tapster Rock⁴, a software vendor producing strategic management software, is a free library that provides exactly this functionality. This library contains methods to get all necessary information like time annotations, relations, and hierarchical decomposition of a *MS-Project* plan.

Asbru plans are loaded by using the internal library *Asbru-Xml*. This library parses a eXtensible Markup Language (XML) plan and creates a data model which contains all information about a plan. This model can be used to read out all plans with regards to their hierarchical structure, furthermore, all necessary time annotations are provided. The usage of *Asbru-Xml* ensures that the prototype will still work even if a newer version of *Asbru* is distributed.

4.1.3 PlanningLines

The prototype must be able to display *PlanningLines* using the intended notation (see Section 3.3.1) as well as all defined design issues of the display (see Section 3.3.2). This includes also the implementation of hierarchical views, logical relations, and a working timescale. Regarding to further development (e.g., view while execution of a plan), all implemented compo-

³For more information about the MPXJ library see <http://mpxj.sourceforge.net> (accessed on May 30th, 2006).

⁴For more information about the Tapster Rock see <http://www.tapsterrock.com> (accessed on May 30th, 2006).

nents must be expandable (e. g., ability of representing *LifeLines* instead of *PlanningLines*).

Going conform with the *InfoVis Pipeline*, abstract data of all activities are stored within a data table. Once an activity is meant to get visible, a data structure which can be rendered on the display is created. The strict separation between abstract data and visual data is often referred as polyolithic design (in contrast to monolithic design where abstract and visual data are stored within one data structure).

Therefore, a *PlanningLine* requires a structure providing all needed visual information. Additionally, a renderer is needed to represent these information. Applying this architecture enables the possibility of completely other ways of painting, therefore the demand of extensibility is met.

4.1.4 Display

The display is the view of the *Information Visualization*. As the concept of *PlanningLines* also contains a notation of representing multiple *PlanningLines* and their relations among each other (see Section 3.3.2), these demands must be fulfilled by the display. It is responsible for an exact and correct representation of all *PlanningLines*.

The display was chosen to be not bounded. The possibility of expanding and collapsing *PlanningLines* would affect the bottom boundary, or, if it would be fixed, the maximum depth of the hierarchy must be calculated beforehand which would violate the on-demand loading concept. Further, all considered toolkits that are eligible to perform the graphical issues works with a not bounded view, therefore, it was obvious to overtake their concept.

However, together with the timescale, this component is the most important element within the visualization. All painting actions (including additional necessary operations like calling a layouter, assigning renderer, etc.), are managed and executed by the display. It needs access to all visual structures and decides when the *Information Visualization* has to be repainted.

Also user interactions have to be maintained by the display. Depending on the requested action, the display has to forward incoming events to an object that performs the requested actions or must handle the events itself. As user interaction typically changes the view, so repainting of the view is necessary whenever a user interacts.

Besides issues given by *PlanningLines*, the display has to refer to the timescale. In fact, ticks represented by the timescale should be continued on the displays' background. This helps users to identify points represented by *PlanningLines*.

Another requirement that helps users to associate a point on the display with a date, is a component called time-cursor. A time-cursor is a simple vertical line in front of the display (also in the foreground of *Planning-*

Lines), and can be moved along the horizontal axis. This line is used to get detailed information about time depending on its position. On the one hand it adjoins to the timescale so users can identify dates visually, on the other hand the exact time of the position is also represented textually on the bottom of the line.

4.1.5 Timescale

As the timescale is the central and most important element within the entire visualization, it is necessary to apply special attention to it. The timescale must be able to represent each possible time interval. Possible intervals can reach from some milliseconds to many years or even more. The interval should be separated by bigger and smaller lines into two successive units (also called granularity, e.g., main spaces represents weeks, small spaces days). Ticks must be annotated whenever possible (depending on the distance to surrounding lines), also the start and end-point of the actual interval must have a clear caption.

As all other graphical elements within the *Information Visualization* are mounted to the timescale, it must be able to provide the associated date for every single pixel and vice versa, whenever a component requests it.

In case of the prototype, the scale itself must calculate the displayed granularities automatically. A user must not take care about changing the units, an algorithm determines the best fitting granularity depending on the represented interval. Also labeling of single ticks has to be calculated by an algorithm, comparing available and needed space for each label.

As users will navigate within the plan, the timescale has to be recalculated very often. But not only the visible interval changes when navigating, also the applied granularity can change. Changing of these means a new calculation of all graphical parts of the scale on the one hand, and a complete repaint on the other hand. Both, calculation and repaint, consume time, therefore it is very important to reduce calculations to a minimum and take care about the runtime of necessary calculations. The challenge in finding a fast algorithm is caused by the fact that dates (and their positions on the screen) cannot be calculated linear as a calendar contains erratic times (e.g., a month can have between 28 and 31 days).

Furthermore, completely repainting should also be reduced to the absolute minimum, as the scale consists of a large amount of graphical elements (single lines and labels).

4.1.6 User Interaction Techniques

To ensure usability, some interaction techniques have to be applied to the application. As this prototype is only a viewing tool, the main focus is paid

toward navigation within the view and controlling the state of hierarchically decomposed tasks.

As a plan can consist of a large amount of tasks with different durations, they may differ strongly (e.g., the main task lasts two years, its smallest subtask only one day). Therefore it is necessary to care about a reasonable representation. If durations differ really strongly, short tasks would even not be visible if the display is adjusted to the longer tasks.

Therefore, to ensure an adequate representation of each *PlanningLine*, zooming should be applied to the view. Zooming ensures that each *PlanningLine* can be viewed in the appropriate temporal context. Zooming should be controlled with the mouse-clicking on any point of the display (right mouse-button) and dragging the mouse up or down. The view will change its zoom factor as long as the mouse is dragged. But not only the view and the size of the underlying tasks change when zooming in or out, also the represented interval of the timescale changes (possibly granularity also changes as this happens automatically).

Also semantic zooming should be applied. As it would be useless to display very small *PlanningLines*, such *PlanningLines* are only represented as a small gray rectangle to show their existence. Only when a user zooms in and the task becomes big enough a representation makes sense, a *PlanningLine* is painted instead.

To move through a plan, another navigation technique is necessary. One possibility would be the usage of scrollbars, but scrollbars on a view that is not bounded are hard to implement and use. So panning is the favorite interaction technique for navigation. The user just has to click on any point in the view, hold the left mouse-button, and drag in any direction. The view itself (and also the timescale) moves along with the mouse drags until the button is released.

As a plan can contain hierarchical decomposition, the user needs a way to expand and collapse tasks. This should be realized with a double-click on a *PlanningLine* or with a single click on the respective symbol indicating the hierarchy.

4.2 Toolkits

Based on the requirements, open-source toolkits that support the development have to be found. Applying toolkits for graphical representation also means to go conform with the architecture of them in the own application. Therefore, the selection of proper toolkits is an important task that should be considered carefully.

4.2.1 Selection of Toolkits

The first chosen toolkit is the MPXJ library to read *MS-Project* files. As there is no available alternative, there was no discussion about using it. Moreover, reading *MS-Project* files is just a small part within the application and only the basic functionality of the library is needed.

The selection of graphical toolkits was a bit harder work, as each toolkit affects the whole application. Particularly, three open-source toolkits seemed to be appropriate for the prototype: *Prefuse*, *InfoVis Toolkit*, and *Piccolo* (for an overview of these see Section 2.3.6).

To ensure a well designed *Information Visualization*, the *InfoVis Pipeline* is the central design pattern. All single variations of data and transformation of these toolkits must be conform to this recommendation, therefore the decision of adequate toolkits was heavily influenced by meeting the demands of the *InfoVis Pipeline* (see Figure 4.1).

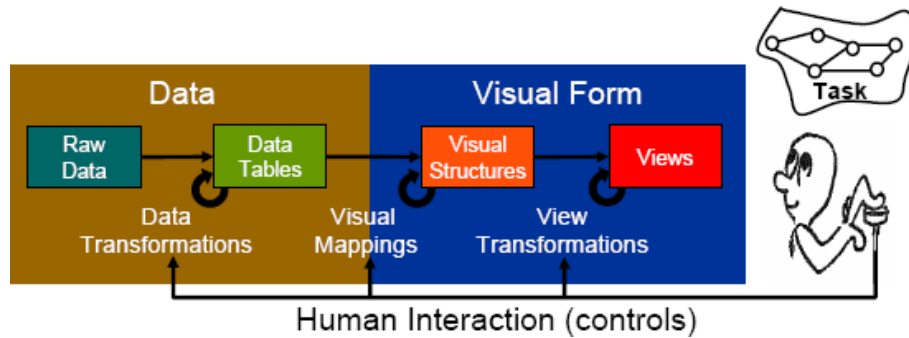


Figure 4.1: Information Visualization with the InfoVis Pipeline [Card et al., 1999].

The first form of data is raw data. In case of the prototype, raw data can be an *Asbru* plan as well as a *MS-Project* plan. In the first step, this raw data is transformed into a data table. A data table is an adequate form to hold information, using indexes also allows to map hierarchical and logical relations among data rows. As only string representations are needed, this data structure saves memory without losing information. Therefore, a data table is an adequate basic structure holding data of the entire plan without wasting resources. Also regarding to expandability and further development it is a proper structure, since it is easily possible to transform every potential data source into a table.

One idea was to use a standard Java *Table* and expand it to meet the demands of the *Information Visualization*. But the *InfoVis Toolkit* is also based on data tables and provides additional implementations of filling them out of a data source. So it was obvious to use this toolkit. Advantages of the *InfoVis Toolkit* table are the spare use of resources and its provided interfaces and mechanism to read data (especially text-based data types like

eXtensible Markup Language (XML) or Comma Separated Values (CSV)) or transform data to a table.

An *Information Visualization* realized with *InfoVis Toolkit* is meant to visualize all data within the table at once, since its display is bound to it. As not the whole plan should be visualized at once, *Prefuse* and *Piccolo* seemed to be more suitable for this purpose. Further, these toolkits provide more possibilities of rendering, animation, and interaction.

Piccolo provides a highly sophisticated view on visual structures. Animations can be applied easily, its camera technique enables different perspectives, and its zooming possibilities (as most attention was set to this issue) are terrific.

However, visual data structures are hold together with other elements like cameras or layers within a scene graph. Furthermore, there is no separation between the visual elements and abstract data itself. This design is called monolithic, that means a base class provides most common functionality for different types of visual items (including abstract data, visual data, painting, etc.). To create own items, just an extension of this base class is necessary [Piccolo, 2006]. On the one hand, this design makes it easy to create visual items, but on the other hand, a lot of non-used functionality may waste performance and memory.

The *Prefuse* architecture forces a clear separation between abstract and visual data. This saves resources and results a clear and understandable source-code without needless elements or functionality. Furthermore, *Prefuse* works with a predefined graph structure that also supports trees (a graph with a unique root). This structure is exactly what was needed to store hierarchical decomposed activities of a plan, therefore it was obvious to use *Prefuse* at least for data storage.

Considerations about combination of *Prefuse* with *Piccolo* were discarded fast as their architectures are too different. First, it seemed that *Prefuse* allows a fast development regarding to data management, but doing the visualization itself would be a real hard work. The monolithic design of *Piccolo* promises quick visual results, *Prefuse* with its polyolithic design needs more time until results are visible.

But the polyolithic design of *Prefuse* provides some advantages regarding to extensibility, especially applying different kinds of graphical representation. Even though working with *Prefuse* is not that easy as *Piccolo* (at least in the beginning), it was realized quickly that *Prefuse* has the power to meet all demands of the prototype. And once the complex architecture of *Prefuse* is conceived, fast results can be realized.

Another important benefit of *Prefuse* is that the toolkit also was designed considering the *InfoVis Pipeline*. This fact makes it easy for the prototype to stay conform with the reference model. Picture 4.2 depicts the parts of the prototype with regards to the *InfoVis Pipeline*.

In contrast to the original concept, an additional data form (therefore

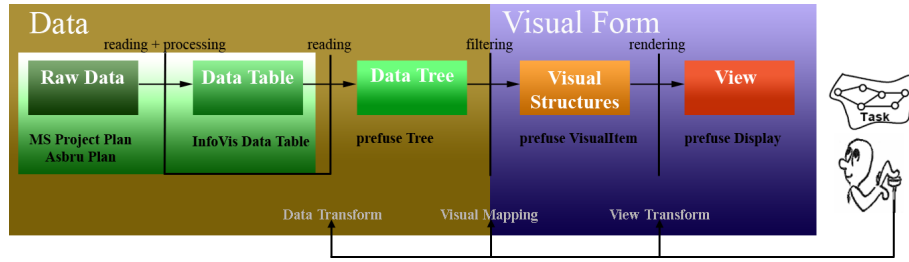


Figure 4.2: Prototype InfoVis Pipeline considering used toolkits.

also an additional transformation) can be seen within the model applied to the prototype (compare with Figure 4.1). *Prefuse* depends on a graph or tree, therefore this data form is indispensable. Although, the table is applied even if it would be possible to create this tree directly out of the raw data. Two reasons legitimate this additional data form:

1. The prototype takes two different kinds of raw data. The table can be seen as a standardization of the input, further development with the aim of supporting other data sources just has to care about filling the table, without having detailed knowledge about the graph structure used by *Prefuse*.
2. As I/O operations waste a lot of time, it is essential to read a plan at once. The data table is a memory saving element to hold the whole plan throughout the runtime of the application. This enables a dynamic implementation of the tree, *Prefuse* nodes with all their functionality are just created when they are definitely needed (first appearance of an activity).

4.2.2 InfoVis Toolkit

A data table is the base of an *Information Visualization* done with the *InfoVis Toolkit*. As only the use of this table is designated within the prototype, the main attention is paid toward this data structure.

A *Table* consists of metadata and named *Columns*. Besides standard usage of a data table, also graphs and trees are supported by defined columns enabling the establishment of relations through indexes. But special behaviors can be also realized easily manually when needed.

Similar to the *Table*, a *Column* also consists of metadata and *Rows*. The metadata allows to specify a special type of *Columns*. A *StringColumn* only consists of strings, an *IntColumns* of integers, etc. A lot of different columns are predefined (see Figure 4.3), extensions are possible. Each *Column* can have empty rows too, which is a very important issue in case of sparse data

(also concerns *PlanningLines* as empty values are supported). But besides standard types like strings and integers, also more complex types like general Java objects can be mapped within a column [Fekete, 2006b].

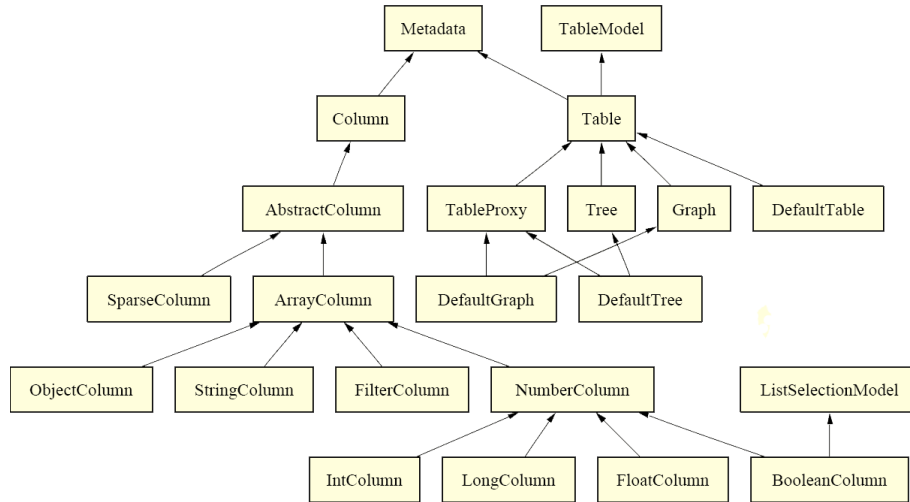


Figure 4.3: Hierarchy of InfoVis Toolkit tables and columns [Fekete, 2006b].

Typically, a table consists of one or more columns. As a column is an array of a given type, it contains indexes, thus the table itself is indexed too. Even if a real row object is missing, it can be generated easily by accessing the values of columns through the indexes.

The *InfoVis Toolkit* also supports changes in tables during runtime. Notifications (events) about changes can be used to apply these to the visualization itself. Also formation of data is supported (e.g., a date can be stored within a *LongColumn*, although dates are returned by the column).

As already mentioned, tables measurably improve the memory footprint and performance compared to other data structures for complex types [Fekete, 2004]. The basic of an *Information Visualization* done with *InfoVis Toolkit* is the table which can access raw data through defined readers and writers. Once a table is loaded, the proper visualization is performed (see Figure 4.4).

The object *Visualization* is bound to the table. Semantic attributes which are defined in associated columns are transformed into visual attributes (e.g., color, size, label, sorting order). Besides the standard visual attributes, specific attributes like coordinates or relations between objects can be defined in separate columns too. Each *Visualization* maintains a set of shapes which again are associated with attributes. Shapes are held within the *ShapeColumn* and are repainted whenever it is necessary. Painting is done by the rendering function [Fekete, 2006b].

The visualization also supports changing of the original data. Whenever

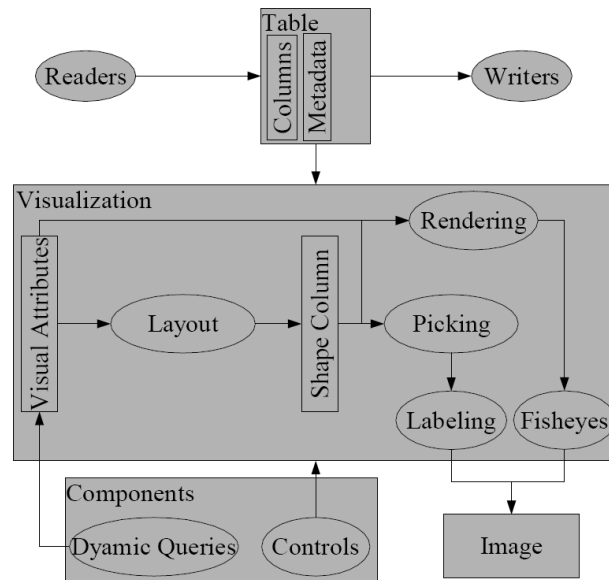


Figure 4.4: Internal structure of the InfoVis Toolkit. Squares represent data structures whereas ellipses represent functions [Fekete, 2004].

the table changes, the *Visualization* is notified and the whole representation is repainted or even recomputed if a visual attribute is affected of the change [Fekete, 2006b].

In *Components* a large set of interaction components are concentrated. Predefined components are combined within own control panels and include sliders to change size or colors of items, but also components for selection or clicking items [Fekete, 2004].

Another possibility of managing the *Information Visualization* are dynamic queries. These are simple expressions used to filter rows of the data table. It is possible to combine more than one filter expression using conjunctions. These queries can also be managed by visual components, such as sliders to interact to the visualization [Fekete, 2004].

InfoVis Toolkit is a toolkit, which pays most attention toward saving resources. Even if there is a measurable advantage in comparison to other toolkits, there are other factors like usability or simplicity too that characterize a good toolkit. The toolkit is very hard to use, at least there is no underlying logic a developer can conceive quickly. But the toolkit provides a lot of examples for standard problems, like scatter plots, etc. These predefined applications are easy to use if no special demands of the *Information Visualization* are needed.

4.2.3 Prefuse

Prefuse was chosen to handle all the graphical stuff of the prototype. Also demanded interaction techniques should be fulfilled with this toolkit, therefore, special attention is set to all given features of the toolkit. With applying *Prefuse* also its architecture must be inherited by the own application.

Like the prototype, *Prefuse* was also developed considering the *InfoVis Pipeline* (see Figure 4.5). Fulfilling the design recommendation of the *InfoVis Pipeline* demands a lot of different classes providing the necessary separation between abstract data, visual data, and views. But once such a polyolithic design is reached, further extensions are easily to perform because of the high data abstraction.

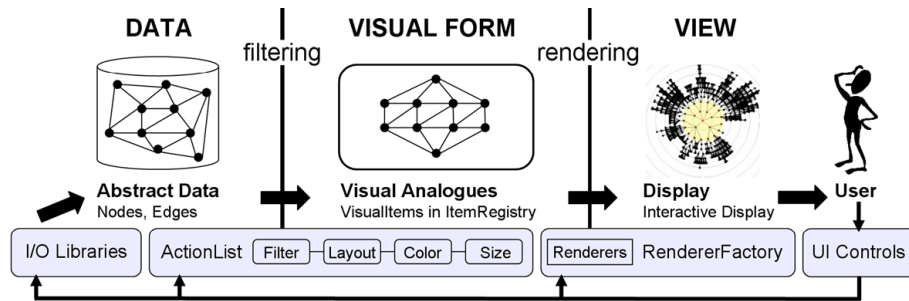


Figure 4.5: Prefuse framework according to the InfoVis Pipeline [Heer et al., 2005].

Even if *Prefuse* provides mechanism for reading raw data, the *InfoVis Toolkit* is responsible for this part in the prototype. The central element in *Prefuse* is the *ItemRegistry*. This container is responsible for holding abstract data and creating visual mappings of these data when needed.

The *ItemRegistry* supports a graph as well as a tree to store abstract data. Internally, the structure is referred as backing graph. The structure is established by using *Nodes* and *Edges* connecting belonging *Nodes*. Furthermore, the *ItemRegistry* also maintains all created visual elements associated with the abstract data. It also provides mechanism to map between abstract and visual data.

A visual entity is called *VisualItem* and contains all necessary information needed for visualization (like location, size, color, etc.). Nodes as well as edges of the graph can be visualized. Furthermore, the *ItemRegistry* also maintains different *Renderers* for each type of *VisualItem*. A *Renderer* implements all necessary painting issues.

The transformation of abstract data to visual analogues is done through filters which reduce the abstract information (*Node* or *Edge*) to visual content (*VisualNode* or *VisualEdge*) [Heer et al., 2005]. The filtering process is equivalent to the visual mapping transformation of the *InfoVis Pipeline*.

But, besides filtering the content itself, also assignments of the resulting visual structures are typically applied together.

Filtering is realized with a list of *Actions* that are stored within a runnable container called *ActionList*. An *ActionList* has a reference to an *ItemRegistry* and is invoked manually in code, periodically as defined, or on users' interaction (e.g., on click on an item).

Typically, a filtering *ActionList* consists of following parts (see Figure 4.5) [Heer, 2004]:

1. *Filter*: This *Action* is responsible to transform abstract data as stored in the backing graph to visual analogues. To do this task, the filter steps through the backing graph and creates a *VisualItem* for each abstract instance. Additionally, garbage collection of not needed items within the *ItemRegistry* is initiated by the filter.
2. *Layout*: This *Action* is responsible for a correct placing of items within the view. Typically graphs or trees are represented within *Prefuse*, so some common layout algorithms (e.g., *ForceDirectedLayout* or *Radial-TreeLayout*) are already predefined in the toolkit. Special algorithms can be easily implemented by extending existing algorithms or writing own ones from scratch. A *Layout* has direct access to the *VisualItem* it should place, so assignments can be performed easily.
3. *Assignment*: These *Actions* set visual attributes like colors, sizes, fonts, etc. This is very useful in common visualizations, where such assignments are often dependent on abstract data.

Generally, the layout *Action* is also an assignment *Action*, but as layout is one of the most important parts within an *Information Visualization*, this *Action* can be seen separately. Additionally to the filter process, also animation of visual attributes like color or size can be realized by using periodical *ActionLists* that are executed whenever a defined duration elapsed.

Once all *VisualItems* are created, they can be rendered onto the *Display*. The *Display* is the graphical component of the *Prefuse* toolkit. As most of the *Prefuse* components, also the *Display* is connected with an *ItemRegistry*. In addition, multiple *Displays* can refer to the same *ItemRegistry*, enabling different views (e.g., see Figure 4.6: detailed view in combination with an overview).

The *Display* is an extended component of the Java Swing *JComponent*, a super class for all painting issues. Therefore, the *Display* has also inherited the whole Java Swing painting architecture including *Double Buffering*. Nevertheless, *Double Buffering* is done manually for some performance reasons and to have more control about it.

Whenever the *Display* has to be repainted, it requests all *VisualItems* of the *ItemRegistry*. As each *VisualItem* has an associated *Renderer* doing

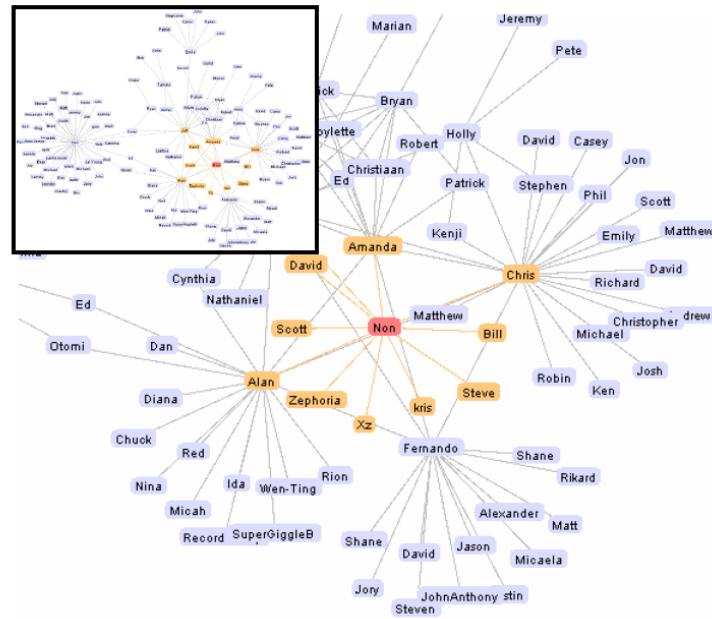


Figure 4.6: Sample of a visualization using a second overview display [Heer, 2004].

the painting, the *Display* just has to paint itself (e.g., the background) and then delegates the painting of each item to the respective *Renderer*. Before painting a *VisualItem*, the respective *Renderer* is asked for the position and maximal area the graphical representation of the item needs. So the *Display* can decide which items have to be painted and which not. Such a bounding-management saves resources, as items that are not visibly anyway on the actual view are not rendered.

Besides predefined *Renderer* that cover often used forms used in a common *Information Visualization*, it is possible to create own *Renderers* and associate them with the *RendererFactory* of the *ItemRegistry*. Following this architecture, different *Renderers* can be applied to items, enabling different graphical representations for example used to enable semantic zooming.

Painting itself uses the standard painting architecture of Java Swing. That means, the Java *Graphics* object of the *Display* is passed through all visual components, which can perform their own paintings using standard Java components like strokes, colors, and fonts. The user is not bound to predefined painting methods of a toolkit, the whole range of possibilities provided by Java can be used. Therefore, *Prefuse* allows developers to create each thinkable graphic to communicate information.

The Java2D library is used to support affine transformations. The graphic context is bound to a matrix, allowing transformations of the whole

Display. That means, besides the logical coordinate system (*absolute coordinates*), there is a virtual one (*view coordinates*). When painting, each point of the *absolute coordinate* system is translated into the *view coordinate* system (see Section 6.1.2).

Using affine transformations provides possibilities like resizing (zooming), translation (panning), or rotation without changing any assignment attributes of visual elements. As conversions between *absolute coordinates* and *view coordinates* are done automatically, a developer usually must not consider the *view coordinate* system. Zooming or panning only change the underlying transformation matrix, through use of automatic routines of Java2D the results are displayed (details about affine transformations can be found in Section 6.5.2).

The Display also supports interaction with visualized items, as the Java *ControlListener* interface is applied. This interface calls defined code whenever mouse or keyboard events arise. Therefore, a user just has to provide own methods collected in a *ControlListener* object to be able to react on interaction events. This can be done with *ActionLists* for example.

Concluding, *Prefuse* is a very powerful toolkit, providing a large set of components and methods a developer needs to build an *Information Visualization*. There are a lot of predefined elements (like renderers, layouts, or controls) that are sufficient for smaller visualizations.

As mentioned earlier, the polyolithic design is not that easy to understand in the beginning of a development, but once conceived, a visualization is easy to realize even if special demands have to be fulfilled. Furthermore, using *Prefuse* forces a clear and well designed architecture of own applications.

Prefuse was created with most attention paid to developers who will finally work with the toolkit. In an evaluation study this goal was approved [Heer et al., 2005]. In the meanwhile, there is also a not so small community using and upgrading the toolkit, which is an argument for its usability.

The next Chapter describes the implemented features of the *Information Visualization* that were realized by using the described toolkits. Also imaginable features that are not implemented yet are discussed.

Chapter 5

Prototype Implementation

As all requirements of the prototype were defined and toolkits were chosen, the implementation started. The result of the implementation is an *Information Visualization* that makes it possible to load *MS-Project* plans as well as *Asbru* plans, and represent these plans using the concept *PlanningLines*. As this prototype is designed to be also part of *CareVis* and not a stand alone application, the executable window that implements this *Information Visualization* is only for testing and presentation purposes. Nevertheless, some features provided by this executable user interface are included in this overview too, as the work is performed by the *Information Visualization* itself.

In Section 5.2 a short user manual of the application is given. There, interaction possibilities and the menu bar is described.

5.1 Features

The prototype is a viewer for *Asbru* plans and *MS-Project* plans in the notation of *PlanningLines*. All design concepts of the charting technique *PlanningLines* are realized. The application uses a sophisticated view providing different navigation techniques. Time is depicted with a timescale on the top of the view that adapts itself depending on the actual view.

5.1.1 Data Sources

Two different types of data sources can be used within the application which are *MS-Project* plans and *Asbru* plans. There are two different file name extensions of *MS-Project* plans: *.mpp and *.mpx. Theoretically, both should be usable, practically only *.mpp plans can be used since the MPXJ library to read *MS-Project* plans has some problems with the *.mpx format.

If indeterminacies of time annotations appear during reading a plan, missing values are calculated automatically if possible. This minimizes spe-

cial constellations of tasks (see Section 3.3.1). As *MS-Project* plans do not contain most of the time annotations at all, also a randomizer can be used to fill these missing values. The randomizer has no benefit for the end user, but it is helpful for presentation of the application.

5.1.2 TimeScale

The timescale maintains itself, no adjustments or configurations have to be done by a user. Time is displayed using two related units, one for the main separations, the other to separate these main intervals again into smaller units. The choice of these units (also called granularity) heavily depends on the actual interval a user views. Therefore, granularity is calculated automatically and also changes automatically when the interval has changed. This ensures an appropriate representation of time regardless how detailed or rough a plan is displayed.

The highest granularity that is used are years, the smallest are milliseconds. Additionally, besides the logical graduations like days or weeks, there are also granularities like “four hours” or “four months”. These additional granularities provide a smooth changing between units and therefore a scale that is always separated in a way a user expects.

Also labeling of the scale itself and units of separations is done automatically. Single ticks get a caption whenever it is suitable and possible. If captions are possible or not depends on available space to the next tick. Furthermore, the actual start and end are also labeled, just as an information about the main granularity that is used at the moment.

5.1.3 Activities of a Plan

There are two different modes to display an activity of a plan, and other two modes how to display hierarchical decomposition. Thus, altogether there are four views that can be applied to a plan (the execution window provides them in a menu to switch between them).

Of course, the preferred view are *PlanningLines*. A *PlanningLine* is displayed as given by the concept (see Section 3). Each *PlanningLine* is labeled and an additional small triangle depicts whether the task is composed of further subtasks that can be expanded. Special constellations that occur because of missing time annotations of a task are also represented, since it is required by the concept (see Section 3.3.1). Furthermore, if EST and LFT are equal, instead of a *PlanningLine* a milestone is represented.

As a preparation of the *Information Visualization* for displaying a plan at execution time, the duration bars are movable between the respective bounds. For demonstration, the moving of duration bars can be done by using a shortcut (*Ctrl*) in combination with the dragging mouse (left button) too.

The next view that can be applied are *LifeLines*. Instead of representing all time attributes of a task, only a single labeled rectangle containing the symbol to expand or collapse children is displayed. The length of the *LifeLine* is given through Earliest Starting Time (EST) and Latest Finishing Time (LFT).

There are two views to display hierarchical decomposition that can be applied to both mentioned representations. The first one just leaves the *PlanningLine* or *LifeLine* when it is expanded. Only the expand/collapse symbol indicates the state of a task. The second way is to depict an expanded task by using a labeled summary bar instead of the original representation. Expanded tasks can be identified easily and children which belong together are separated clearly.

Independent of the representation of a task, a tooltip always provides the most important information about it. Furthermore, if a task is too short for an adequate representation as *PlanningLine* or *LifeLine*, only a small labeled gray bar indicates the existence of it. Once the resolution of time is high enough (zoom in) so that a representation as *PlanningLine* or *LifeLine* makes sense, the proper representation is done automatically.

Concluding, logical relations between tasks are displayed using arrows that connect related tasks and depict also the direction and kind of relation (Start/Start or End/Start, see also Section 3.3.2). A layout algorithm tries to find the best possible arrangement of linked activities.

5.1.4 View

The view provides the layouted sight onto a whole plan, and consists of at least one activity. If a plan is loaded, the view, and therefore the timescale too, is initialized in a way that the whole time interval a plan lasts is visible. The first task (root task) is placed centered in the top of the view's area. The free space to the left and right is described by a factor, which can be chosen programmatically depending on the application the *Information Visualization* runs in.

Furthermore, the view is responsible for navigation within a plan. Therefore, two different techniques are applied. Zooming is done by dragging the mouse with pressed right button over the view. Besides changing the size of the displayed items, also the represented interval (and maybe the displayed granularity) of the timescale changes along the zoom factor. Zooming is only applied in the horizontal direction, that means that the height of a single *PlanningLine* never changes (the same applies to vertical distances between tasks too). Vertical stretching would lead to a confuse and unclear view as *PlanningLines* and vertical spaces would get too large. This would also lead to a loss of communicated information as a user cannot recognize coherencies any more at a certain zoom level.

Panning is done the same way as zooming, but instead of the right mouse-

button the left one must be pressed. Similar to zooming, the timescale adapts itself to the actual area the user watches. But in this case, the granularity cannot change as the relative time interval of the scale stays always the same when panning.

Small buttons on the top of the timescale allow quick navigation. One resets previously done zooming and panning, so the whole plan can be seen in its original size and at its original position. Other buttons pan to the beginning or ending of a plan.

Single ticks represented by the timescale are continued on the view's background. This background separation provides users a better identification of time facets of overlying tasks. Furthermore, the view provides a time-cursor, a single red line displaying the actual time of its positions. This time-cursor can be moved by the user (dragging with left mouse-button) along the horizontal axis of the view to get the exact time of a position he is interested in.

5.1.5 Screenshots

This Section shows some screenshots of the implemented prototype. All pictures depict the same *MS-Project* sample project which shows most of the implemented notation.

Figure 5.1 shows the timescale of the prototype. At the moment a granularity of weeks is selected, further, each week is separated into single days. The small buttons on the top are used for navigation, the middle one resets the zoom factor and centers the plan, the left one moves the plan to the upper left corner, the right one moves it to the upper right corner.



Figure 5.1: The timescale of the prototype.

Figure 5.2 illustrates the sample project. The gray tasks illustrates tasks where no durations are given. All tasks are children of the top most task “sampleproject” which is expanded and therefore shown as summary bar. Further, all task are related to others which is illustrated by the blue arrows. The red line is the moveable time-cursor.

Figure 5.3 illustrates the same project except that a higher zoom factor is used. As the actual granularity has switched from month to weeks, the background also highlights the weekends.

Figure 5.4 shows the four different views of the sample project. A task can be presented as *PlanningLine* or *LifeLine*, further, hierarchical decomposition can be represented with a summary bar by leaving the original

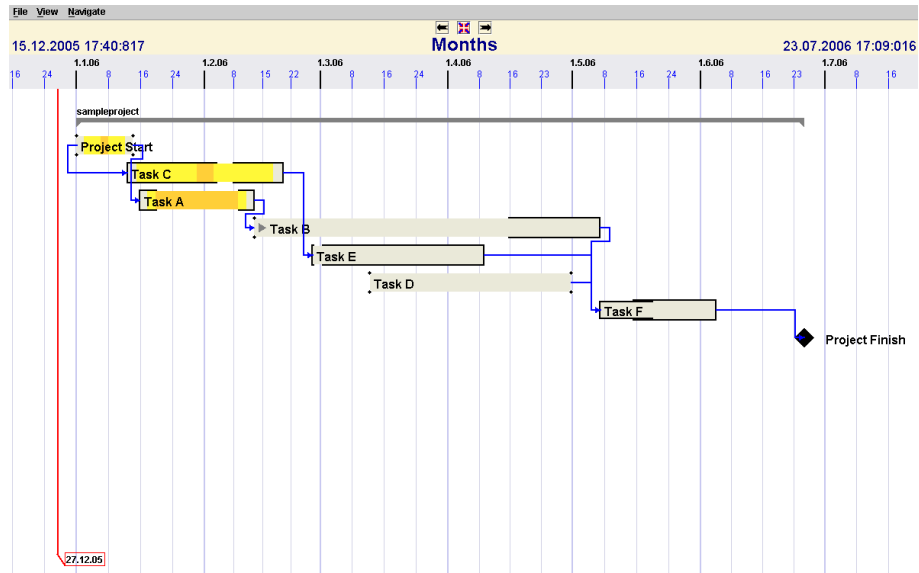


Figure 5.2: The prototype shows a MS-Project sample project.

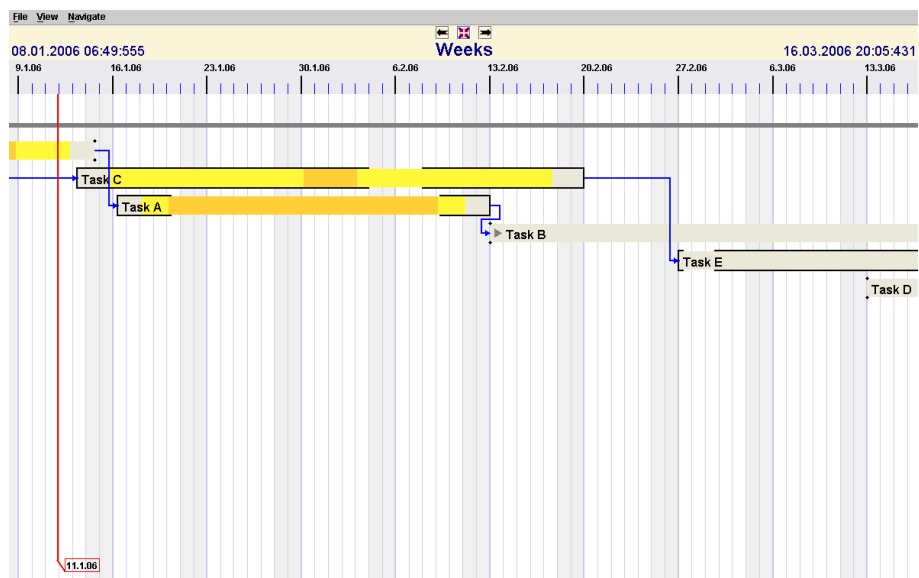


Figure 5.3: The same sample project shown with a higher zoom factor.

visual structure (only the expanding-symbol indicates the expand/collapse state).

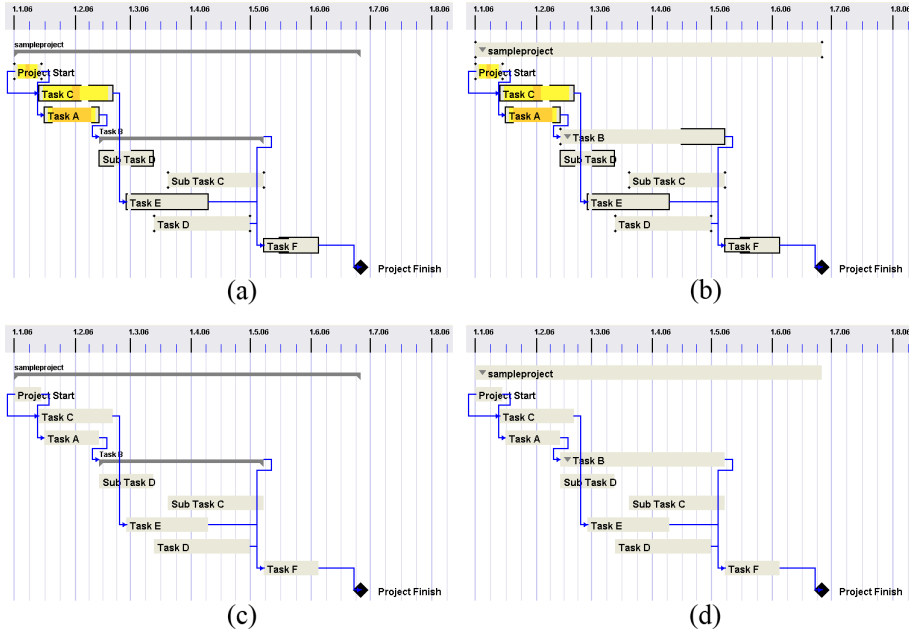


Figure 5.4: This picture illustrates the different views of the same plan. (a) shows hierarchical PlanningLines which uses a summary bar to represent an expanded task, (b) only shows PlanningLines, (c) shows hierarchical LifeLines, and (d) shows LifeLines without summary bars.

5.2 Manual

The whole *Information Visualization* is controlled by using the mouse. Expanding or collapsing of nodes is done with clicking on the respective symbol of a *PlanningLine* or performing a double-click on any point of a *PlanningLine*. Panning and zooming is done by holding a mouse-button and dragging the mouse in any direction. The left key is used for panning, the right one for zooming.

To move the duration bars of a single *PlanningLine*, the *Ctrl*-key in combination with the left mouse-button must be pressed while moving the mouse on a bar to the left or right.

Customized tooltips of tasks are shown whenever the mouse-cursor is left on a *PlanningLine* or *LifeLine* for a short while.

The menu provided by the execution window provides some functionality too. The “File” menu is used to open plans or quit the application. The different views that can be applied to a plan can be selected in the “View”

menu. Furthermore, the functionalities of the navigation buttons of the *TimeScale* are defined in the “Navigate” menu again:

- **File:**
 - **Open:** User can choose an *Asbru* or *MS-Project* plan to display. This function is also available when a plan is displayed already. The actual one is just replaced by the new one.
 - **Open with Randomizer:** Instead of the “Open” entry, this dialog can be used to open a *MS-Project* plan (*Asbru* is not possible). All time annotations may be generated randomly, already defined ones may be deleted too. So, also special constellations are generated.
 - **Quit:** Exits the application.
- **View:**
 - **LifeLines:** Only *LifeLines* are shown. Hierarchical decompositions are also shown using *LifeLines* instead of summary bars.
 - **Hierarchical LifeLines:** *LifeLines* are shown, hierarchical decompositions are depicted using summary bars.
 - **PlanningLines:** Only *PlanningLines* are shown. Hierarchical decompositions are also shown using *PlanningLines* instead of summary bars.
 - **Hierarchical PlanningLines:** *PlanningLines* are shown, hierarchical decompositions are depicted using summary bars.
- **Navigate:**
 - **View All:** The plan is placed at its initial position (nearly centered) and the zoom factor is reseted. Depending on depth of expanded tasks, the whole plan or at least the upper area of it is shown at once.
 - **Go To Start:** Pans the view to the upper left area of the plan. The zoom factor stays at it is.
 - **Go To End:** Pans the view to the upper right area of the plan. The zoom factor stays at it is.

5.3 Not Implemented Features

The following features are thinkable for further implementation, but not realized yet:

- Controlling the *Information Visualization* using the keyboard and shortcuts.
- Second view displaying an overview of the whole plan (*Bird View*) used for navigating within the original view.
- *Fisheye view* for displaying details of a region of special interest.
- Providing all temporal attributes textually too.
- Change of visual attributes like color, font, etc. at runtime.
- Providing dynamic creation of additional time-cursors, so users can use multiple of them to mark positions of interest.
- Possibility of changing the visibility state of items manually. This option would allow to watch only a set of items a user is interested in.

The prototype was implemented as a viewer of plans. But it is also imaginable to use it for plan-authoring or monitoring a plan while execution. Preparing it for execution should not be that difficult, but authoring of plans would cause a lot of work, as the whole logic of a plan and the functionality to create tasks would have to be implemented.

The next chapter describes the architecture of the prototype. It is not a detailed description of the source code, rather technical concepts and the idea behind them are outlined. Further information about the toolkits, basic environment, but also interaction techniques can be found there too.

Chapter 6

Prototype Architecture

This Section gives an overview about the architecture and technical details of the prototype. Throughout this chapter *UML*¹ *class diagrams* are used to facilitate the understanding of architectural details (see also UML Notation, Appendix B). Note that following UML diagrams are not exhaustive, they are just used to communicate the most important coherencies as this chapter is not thought as a complete documentation of source code.

Additionally, some general diagrams or images following no explicit specification or notation are taken whenever it is suitable (see Section “*the many ways of spreading information*” 2.3).

Generally, the entire architecture was heavily influenced by the architecture of *Prefuse* as nearly all visual components are implemented using this toolkit and its framework. Furthermore, all data transformations (from raw data until a complete view) are conform with the *InfoVis Pipeline* (see Section 2.3.4). Therefore, describing the architecture of the prototype also follows this recommendation. Figure 6.1 depicts the most important objects used within the prototype regarding the *InfoVis Pipeline*.

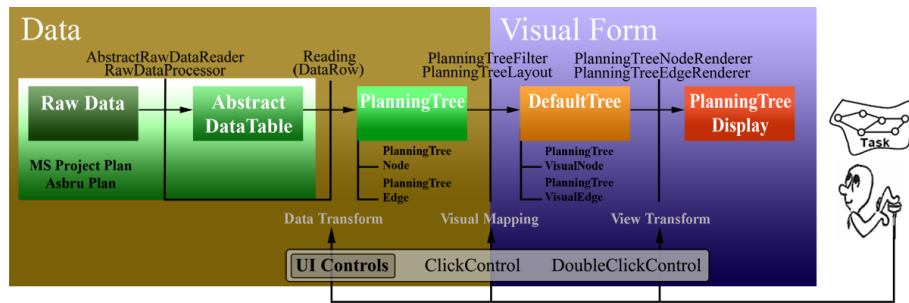


Figure 6.1: Prototype InfoVis Pipeline considering used objects within the application.

¹Unified Modeling Language

6.1 General

6.1.1 Package Structure

Figure 6.2 shows the general package structure (also known as namespaces [Wikipedia, the free encyclopedia, 2006b]) of the implemented prototype:

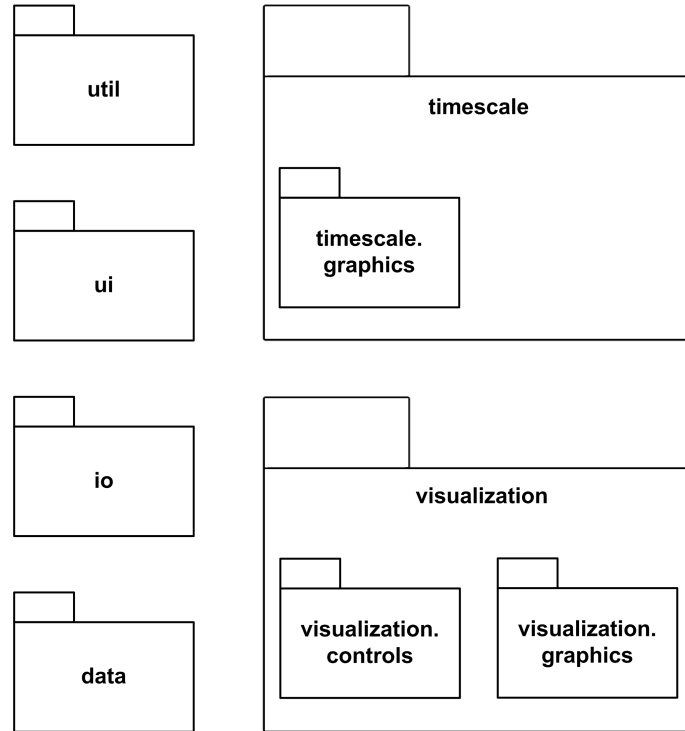


Figure 6.2: Package structure.

- **ui:** Consists of user interface classes. Also the runnable top-level UI container, that is used to demonstrate the proper *Information Visualization*, can be found in this package.
- **util:** General helper classes used throughout the whole implementation. Also interfaces and a static class doing conversions between the different coordinate systems can be found in this package.
- **io:** Namespace for all relevant classes used for input and output issues. Also a factory managing different data reader can be found here.
- **data:** This package contains the data table to hold the abstract data read from a file. Furthermore, also a class manipulating single data rows in case of indeterminacy is in this package.

- **timescale:** The static timescale.
- **timescale.graphics:** Contains necessary classes to paint the timescale, like ticks or labels.
- **visualization:** This package contains all classes needed by the *Pre-fuse* toolkit like container to hold abstract and visual data, layout mechanism, and renderers.
- **visualization.controls:** This sub-package provides classes that handle all relevant user interactions.
- **visualization.graphics:** Holds graphical elements of the application. All visual items to represent a *PlanningLine* are composed with objects of this package.

6.1.2 Coordinate Systems

Java maintains two different coordinate systems [Sun Microsystems, 2001]:

- “*User space is a device-independent, logical coordinate system. Applications use this coordinate system exclusively; all geometries passed into Java 2D rendering routines are specified in user space.*”
- “*Device space is a device-dependent coordinate system that varies according to the target rendering device.*”

From now on, user space coordinates are called *absolute coordinates* and device coordinates are called *view coordinates*. Besides automatically done conversions to transform *absolute coordinates* to *view coordinates* (often performed by platform device driver [Sun Microsystems, 2001]), there is another important transformation mechanism which can be influenced manually: the *Display*’s transformation matrix. This matrix is used to perform user interactions like panning and zooming without changing *absolute coordinates*.

Generally, all coordinates with regard to time within this application are *absolute coordinates*, conversions are done by the *Graphics2D* object automatically when rendering the *Display*. Whenever a user does some interaction that changes the alignment of the view, technically only the matrix is concerned of this change.

But the *TimeScale* also has access to the transformation matrix and provides all calculations in both coordinates systems. Furthermore, some visual objects also calculate points manually by using the matrix. Therefore, a static helper class (*Coordinates*, see Figure 6.3) provides functionality to make conversions manually.

This is necessary since some rendering routines have to be executed manually, as automatically routing (and therefore automatically conversion)

would cause deformations because zooming is only applied in the horizontal direction. But visual elements of Java expects equal zooming in both directions, they do not bother about the fact that one direction is not touched.

A simple vertical line, for example, gets the thicker the higher the horizontal zoom factor is, it is stretched. The thicker line would be suitable if zooming is performed in both directions, as the whole object would get larger and therefore the lines must also be adapted. But in the case that vertical zooming is disabled, the thicker line is not adequate as the proportions to the rest of the element are not suitable anymore.

Thus, concerned visual elements disable the automatic routine of rendering and have to perform conversions between the coordinate systems manually (see Section 6.5.2). In this case, this deformation concerns vertical lines of interval caps, expanding-symbols of a *PlanningLine*, and labels.

Coordinates
<pre> +getAbsoluteX(in t : AffineTransform, in x : double) : double +getAbsoluteY(in t : AffineTransform, in y : double) : double +getAbsoluteWidth(in t : AffineTransform, in w : double) : double +getAbsoluteHeight(in t : AffineTransform, in h : double) : double +getViewX(in t : AffineTransform, in x : double) : double +getViewY(in t : AffineTransform, in y : double) : double +getViewWidth(in t : AffineTransform, in w : double) : double +getViewHeight(in t : AffineTransform, in h : double) : double </pre>

Figure 6.3: Static coordinate conversions.

Furthermore, all coordinates in both systems are stored in double precision. As each *absolute* x-coordinate is associated with an exact date from the beginning on, this is necessary. Otherwise, zooming would cause wrong *view coordinates* because of ignored decimal places, also captions of tasks or the timescale would be false.

Nevertheless, a *view coordinate* is automatically rounded when it is transformed to a *screen coordinate*. That means, a single screen point (pixel) represents a time interval of indefinite length (depends on the actual granularity) and there is no possibility to calculate it back to an exact *view* or *absolute coordinate*.

6.1.3 Notes on Prefuse

Generally, all implemented elements of *Prefuse* are initialized, stored, and maintained by the class *ItemRegistry*. The *ItemRegistry* is the core element of a visualization done with *Prefuse* [Heer, 2004]. It is responsible to hold items containing abstract data (*Nodes* or *Edges*) within a graph structure and create visual analogues (*VisualNodes*) to abstract items using *Renderers* that are maintained in a factory. Furthermore, the registry also maintains the mapping between abstract and visual data, rendering queues of all visualized items, and references to all *Displays* that work with the content of

this registry (see also Section 4.2.3).

Figure 6.4 illustrates the most important classes and their relations to the *ItemRegistry*.

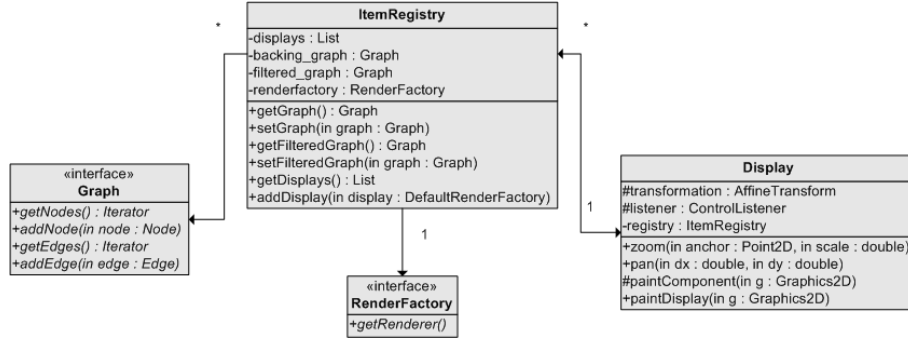


Figure 6.4: UML diagram of the *ItemRegistry*.

6.1.4 Interfaces

There are two important interfaces used within the prototype. The *PlanningInterval* interface is used to exchange abstract data that contains temporal facts. Implementations must provide methods to access and change the respective temporal attributes. The *PaintingObject* is used for rendering issues, implementations must provide assignment information (typically regarding to another *PaintingObject*) and a paint method that can render the object on a *Graphics2D* element (see Figure 6.5).

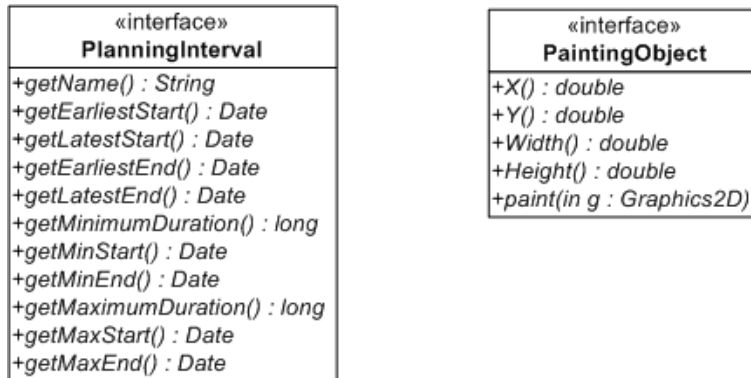


Figure 6.5: Interfaces of the prototype.

6.2 Data

This Section outlines the transformation of raw data to abstract structures that are adequate for later visualizing.

The *AbstractDataTable* is an extension of the *InfoVis Toolkit DefaultTable* and used to store the data of the whole plan. It is the only *InfoVis Toolkit* element used within the application. No other functionality was used of *InfoVis Toolkit*, but the table could be used as base for an own visualization done with *InfoVis Toolkit*.

Prefuse is used to visualize activities. It is important to understand the polyolithic structure of *Prefuse*. For each visual element there also exists a second element containing only the abstract data. Both, abstract and visual objects, are maintained by the predefined *ItemRegistry* that stores them in two separated trees.

6.2.1 Reading Raw Data

Both, *MS-Project* and *Asbru* plans are read using own readers. The *AbstractRawDataReader* defines the functionality of a reader. This class was chosen to be abstract, as it contains some functionality that should be equal for each possible reader (e.g., filling the table with rows).

The task of a reader is to read a given file, extract all single tasks, and write them into the *InfoVis Toolkit* data table. As the *AbstractDataTable* consists only of string representations of data, a reader has to allocate a unique ID so that relations (hierarchical and logical) can be established between the tasks. Further, a reader has to take care about setting the root task (ID 0) which contains at least an EST and LFT. The root task must cover all possible time-points of subtasks.

DataRow

The object *DataRow* stores all temporal attributes of a task and some additional information like name, children, parent, and logical relations. This row also implements the interface *PlanningInterval* to provide functionality to access and set temporal values.

As the *InfoVis Toolkit* does only support indexed columns, this object was developed to fill this gap. Each *DataRow* has a unique ID which is used to request respective data of each column.

The *DataRow* is used to fill the data table, but it is also returned by the table when *Prefuse* needs access to the abstract data. Also implemented abstract data structures used together with *Prefuse* consist basically of a *DataRow*.

A *DataRow* holds the same information as a single row within the *AbstractDataTable*, but instead of a string representation the proper types

are used, e. g. , EST is stored as a string within the table and as Date in the *DataRow*.

6.2.2 Data Preprocessing

Before the data is finally stored in the data table, a static object called *RawDataProcessor* tries to fill temporal indeterminacies within a row (see Section 3.3.1). Therefore, this object searches for indeterminacies within the set of attributes of a task and tries to calculate missing values. This is done to avoid special constellations of *PlanningLines* when missing values can be extracted manually. Missing values can be calculated by considering the constraints of temporal attributes (see Section 3.3.1).

6.2.3 Data Table

The implemented *InfoVis Toolkit* table stores data of all tasks throughout the lifetime of the application ². The *AbstractDataTable* consists of a *IntColumn* for the ID, a *StringColumn* to store the name, several *DateColumns* and *LongColumns* representing all time annotations of a task, and several *ObjectColumns* mapping relationships to other tasks. Relationships are mapped using the unique ID of the tasks which are stored in *Vectors*.

Relations can be hierarchical (*Parent* and *Children* columns) and logical (only successors are mapped in the columns *StartSuccessor* and *EndSuccessor*). Additionally, also the neighbors of all tasks are mapped as *Brothers*.

6.2.4 Reading Data Table

Data of each task is accessed through its unique ID. The ID 0 is the root task of a plan, further tasks are extracted from the children of the root. The ID is also the number of the row within the table.

When retrieving data from the table, a *DataRow* is created again out of a row. All strings stored in the table are casted back to their original type, so further objects can work with the abstract information.

Rows are only accessed when the *Prefuse PlanningTree* creates its own abstract structure of a task for later visualizing. The received *DataRow* is also used as container within the nodes of *Prefuse*.

6.2.5 PlanningTree

Once the data table is loaded completely, a *Prefuse* tree is initialized. The *PlanningTree* is an extension of the predefined *DefaultTree* of the toolkit.

²The latest *prefuse* version contains its own implementation of a data table. When upgrading the prototype to this newer version, the usage of this table may be thinkable.

Basically, this tree holds different entities in a hierarchical form and provides multiple ways of accessing them.

It is important to note that all elements within this tree are only abstract information. None of these objects contains any data needed for the proper visualization, but the abstract information are used to create visual data later.

The *PlanningTree* is initialized with a reference to the already created *AbstractDataTable*. Single entities stored in the tree are objects of the type *PlanningTreeNode*, *PlanningTreeEdge*, and *DefaultEdge*.

To save resources, the *PlanningTree* is only filled on demand. That means, when initializing the tree, only the root node is created (and also visualized automatically). The root node represents the lowest level of the hierarchy within a plan, its duration stands for the duration of the whole plan. Therefore, when the *Information Visualization* is rendered the first time, only this root node is visible. Further nodes are only created on users' interaction.

Technically, each *PlanningTreeNode* knows if it has any children even if they are not loaded yet. This information is provided by the underlying *DataRow* as *children*. The existence of children also affects the visual representation, as a *PlanningLine* is marked as expandable.

Whenever an already visualized item is expanded by any user interaction, the *PlanningTree* is asked if the children of the task are already loaded. If not, the tree extracts information about children of the actual *PlanningTreeNode*, retrieves the abstract data of the underlying data table, creates the respective *PlanningTreeNodes* and *PlanningTreeEdges*, and inserts them into the own data structure.

Once a node is created, it stays in the tree to ensure further access to it (details about expanding or collapsing a task can be found in Section 6.4.1).

PlanningTreeNode and PlanningTreeEdge

As mentioned earlier, *PlanningTreeNodes* and *PlanningTreeEdges* are integral parts of the *PlanningTree*. The tree also consists of *DefaultEdges* used to map the hierarchy, but these entities are never visualized within the application. *PlanningTreeNodes* are connected among each other using this type of edges. Also navigation through the structure of the tree uses *DefaultEdges*, but these mechanisms are provided automatically by *Prefuse*, so they are not outlined here.

A *PlanningTreeNode* is the abstract representation of a task and is an extension of the *DefaultTreeNode*. It is the basic structure which can be transformed into a visual structure when a task is visualized (e.g., dates are translated into x-coordinates, the name is used for the label, etc.).

A *PlanningTreeEdge* is the abstract representation of a logical relation between two tasks. It is an extension of the *DefaultEdge*, but in contrast

edges of this type, *PlanningTreeEdges* are rendered on the view (see Section 3.3.2).

Each *PlanningTreeNode* contains an associate *DataRow* of the data table containing all information needed. As the *polylithic* design of *Prefuse* forces, a *PlanningTreeNode* does not contain any visual information, it is just a container that supports typical functionality of a graph node and stores abstract data. Visual information is created later when filtering the abstract nodes (see Section 6.4.1).

As a *DataRow*, a *PlanningTreeNode* implements the interface *PlanningInterval* helping to spread temporal data of a task. This is obtained by storing the associated *DataRow* within the *Node*.

A *PlanningTreeEdge* consists of two *PlanningTreeNodes*, a predecessor and a successor. Further, a flag indicates the type of relation (Start/Start or End/Start), as the visual analogues of edges must consider the direction graphically. The visual representation of a *PlanningTreeEdge* is also created when this tree is filtered.

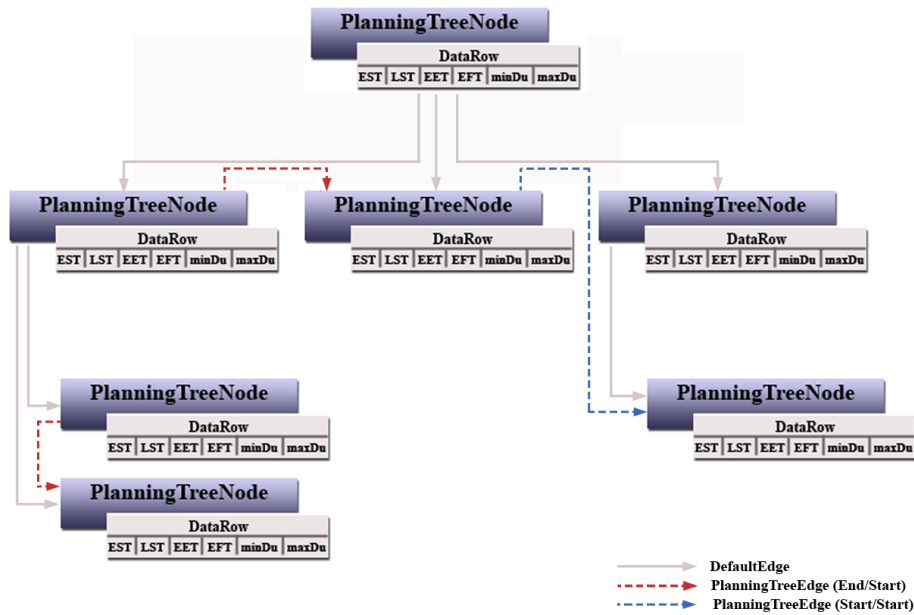


Figure 6.6: Sample of a *PlanningTree* containing Nodes and Edges.

6.3 TimeScale

As the *TimeScale* is the central element within the *Information Visualization*, this element is explained first. The *TimeScale* is a static class and responsible for:

1. Providing an associate date for each horizontal point (x-coordinate) of the view.
2. Providing an associated x-coordinate for each date.
3. Calculating the best fitting units a time interval is separated in (granularity).
4. Providing a visual object of the actual scale.

The *TimeScale* works with all time zones and considers all calendar anomalies like leap years or daylight savings. The highest resolution (granularity) the scale can map are milliseconds, the lowest are years. However, other resolutions than the predefined can be added easily.

6.3.1 General

The *TimeScale* is initialized with the start and end-date of the whole plan it is created for. A factor describes the ratio³ of the maximum span the *TimeScale* can represent compared to the time interval the whole plan lasts (e.g., the ratio is two and a plan lasts four months overall: That means that the maximum span the *TimeScale* can represent is eight months). The resulting granularity of this span is the lowest resolution the scale handles, only zoom in is possible. This restriction would not be necessary, but, it is useful as lower resolutions are senseless because the whole plan has sufficient space within this span.

This ratio is also considered when calculating the aligned start and end-date of the scale when it is visible the first time (see also Granularity, Section 6.3.2). For example, the four month plan starts on May 1, 2006 and ends on September 1, 2006. Considering the ratio and a granularity of “months”, the actual start of the scale is the March 1, 2006 and the end is the November 1, 2006. These dates only describe the actual span of the *TimeScale* when the plan is visualized the first time. Nevertheless, the scale can also display times before or after the given interval.

However, the calculated start-date is used as reference-point for later calculations. In fact, these dates define the *absolute coordinate system* (see Section 6.1.2). The start-time is the *absolute* point 0, the end-time the *absolute* point given by the width.

Once the start and end-dates of the *TimeScale* are set, some factors are calculated that allow translations from a x-coordinate to a date and vice versa. As the precision of these calculations must meet one pixel, the width

³As all dates of the scale are aligned depending on the calculated granularity, the ratio is not quite exact. In reality, the maximum span can be a bit larger than given by this factor.

of the view is used to calculate these factors (spans are always given in milliseconds):

$$TimeFactor = \frac{Width}{MaximumSpan}$$

$$PixelFactor = \frac{MaximumSpan}{Width}$$

Of course, only one of them is really necessary, but further calculations are more clear using both. These factors in combination with the exact start-date are enough to calculate *absolute coordinates* on a given date and dates on a given *absolute* x-coordinate:

$$x = (date - ScaleStart) * TimeFactor$$

$$date = ScaleStart + x * PixelFactor$$

Note, that above mentioned time-points and factors are calculated considering *absolute coordinates*. That means, independent of the actual view, these values and assigned calculations are constant for the whole lifetime of the *TimeScale*.

But the *TimeScale* also deals with *view coordinates*, therefore, it holds a reference to the affine transformation matrix of the *Display*. This is needed to provide above mentioned calculations considering both coordinate systems, on the one hand, but also to be able to calculate a graphical representation, on the other hand. Representation depends on zoom factor and translations done with panning and the actual granularity.

6.3.2 Granularity

Milliseconds is the highest possible resolution the *TimeScale* can represent, further, all calculations also consider milliseconds. But, as the visual representation is granulated into logical units (e. g. , days or years) the *TimeScale* must provide translations to other resolutions than milliseconds (see also Figure 6.7).

The static class *DisplayGranularity* is a helper class of the *TimeScale* which contains all possible granularities. At the moment, all logical granularities from milliseconds to years are predefined, further, some other granularities like “five minutes”, “four hours”, or “four months” are included. As the granularity is automatically adjusted to the actual span (depends on zoom factor), these additional granularities take care of an adequate view where the spaces between ticks are neither too large nor too small.

The switching between granularities is also maintained by the class *DisplayGranularity*. Whenever the zoom factor of the view changes, the actual span represented by the *TimeScale* is calculated again. Based on this span,

the *DisplayGranularity* determines if a higher or lower granularity would be more suitable than the last one used.

Furthermore, this class also provides the functionality to align a date to a certain granularity. Aligning works in both directions. Assuming an actual granularity of “months”, July 11, 2006 may be aligned to July 1, 2006 or August 1, 2006. Aligning always considers the next position of a tick, in case of months, ticks are always set on the first day of the month at 12:00 AM. The aligning process starts with milliseconds and aligns stepwise the date for each granularity in the demanded direction until the certain granularity is reached (see also Table 6.1).

6.3.3 Graphical Representation

As user interactions force a repaint of the whole *Information Visualization*, the *TimeScale* must also provide an actual representation of itself. Graphically, the *TimeScale* consists of lot of small lines (main-ticks and small-ticks between the main-ticks) that separate the displayed interval into some logical pieces (given by the granularity) and a lot of labels (see Figure 6.7). All these elements are combined in the class *Scale*. Like most of the graphical elements within this application, also the *Scale* is composed of visual elements (*MainTicks* and *SmallTicks*) that implement the *PaintingObject* interface.



Figure 6.7: Graphical representation of the *TimeScale*.

The *Scale* is held by the *TimeScale*. Whenever the application asks for an actual representation, the *TimeScale* returns its *Scale*. If the view has not changed since last time of repainting, the last calculated *Scale* is returned, otherwise the *Scale* must be recalculated.

Recalculating of the *Scale* is only necessary if the transformation matrix has changed (through zooming or panning) or the width of it has changed (resize of the window). As the actual granularity of the scale may change when zooming or resizing the view, firstly, the *DisplayGranularity* determines if there is a more adequate granularity (see Section 6.3.2).

The actual granularity is the base for all further calculations as the time spans between *Ticks* are given through it. As dates cannot be calculated linear, the Java *Calendar* class is used for this purpose. This class considers all anomalies like leap years or daylight savings, further, the class handles all common time zones. Therefore, a defined span is not given in milliseconds, instead, the granularity of the *Calendar* class is used. Additionally, a

Granularity	Aligned Down	Aligned Up
Milliseconds	Jul 11, 2006 02:07:34.565 PM	Jul 11, 2006 02:07:34.565 PM
250 Milliseconds	Jul 11, 2006 02:07:34.500 PM	Jul 11, 2006 02:07:34.750 PM
Seconds	Jul 11, 2006 02:07:34.000 PM	Jul 11, 2006 02:07:35.000 PM
Five Seconds	Jul 11, 2006 02:07:30.000 PM	Jul 11, 2006 02:07:35.000 PM
Fifteen Seconds	Jul 11, 2006 02:07:30.000 PM	Jul 11, 2006 02:07:45.000 PM
Minutes	Jul 11, 2006 02:07:00.000 PM	Jul 11, 2006 02:08:00.000 PM
Five Minutes	Jul 11, 2006 02:05:00.000 PM	Jul 11, 2006 02:10:00.000 PM
Fifteen Minutes	Jul 11, 2006 02:00:00.000 PM	Jul 11, 2006 02:15:00.000 PM
Hours	Jul 11, 2006 02:00:00.000 PM	Jul 11, 2006 03:00:00.000 PM
Four Hours	Jul 11, 2006 12:00:00.000 PM	Jul 11, 2006 04:00:00.000 PM
Days	Jul 11, 2006 12:00:00.000 AM	Jul 12, 2006 12:00:00.000 AM
Weeks	Jul 09, 2006 12:00:00.000 AM	Jul 16, 2006 12:00:00.000 AM
Months	Jul 01, 2006 12:00:00.000 AM	Aug 01, 2006 12:00:00.000 AM
Four Months	May 01, 2006 12:00:00.000 AM	Sep 01, 2006 12:00:00.000 AM
Years	Jan 01, 2006 12:00:00.000 AM	Jan 01, 2007 12:00:00.000 AM

Table 6.1: The table illustrates how the date July 11, 2006 02:07:34.565 PM is aligned depending direction and granularity.

multiplier may be used for intermediate granularities (e.g., “five minutes” or “four hours”).

A *Tick* is defined by an exact *view coordinate*. These positions are calculated as follows:

1. Determine the exact earliest date of the *TimeScale*. This is given through the *view coordinate* 0.
2. Align down this date to the actual granularity. The associated *view coordinate* of this aligned date is negative or 0. This coordinate is the position of the first *MainTick* even though it is not visible. Based on the aligned date, further dates which are conform to the granularity are calculated.
3. The respective next date is determined by applying the defined time span of the granularity to the previous date (these calculations are performed through the *Calendar* class, so all anomalies are considered). For each calculated date a *Tick* is created and initialized with the associate *view coordinate*.

Once all ticks are initialized, the *Scale* is ready for rendering. As each *Tick* provides its own painting method, the *Scale* just has to call them sequentially when repainting. Furthermore, the painting method of the *Scale* itself must also paint additional labels for each tick whenever there is enough space.

6.3.4 TimeScaleLayer

The prototype has got an own Java *JComponent* *TimeScaleLayer* that holds and manages the graphical representation of the *TimeScale*. Like all Java Abstract Window Toolkit (AWT) components, the form is repainted automatically on events like resizing, moving, or getting the focus.

But the layer is also bound to the entire view of the visualization, the *PlanningLinesDisplay*. Whenever the *Display* has to be repainted, it checks if the *TimeScale* has to be repainted too. In this case, the paint method of the *TimeScaleLayer* is called directly.

Besides the *Scale*, also some additional elements are painted, like colored background and labels indicating the used granularity and the actual start and end of the *TimeScale*.

Even if a Java *JComponent* supports automatic *Double Buffering*, the *TimeScaleLayer* performs this task manually. It holds the offscreen picture of itself in memory to avoid complete repainting whenever it is not necessary. That means, as the *TimeScale* calculates the *Scale* only if horizontal alignment has changed, the *TimeScaleLayer* also paints the *Scale* only once on its offscreen. Further paintings of the *TimeScale* are performed using this

offscreen picture to ensure fast painting and avoid dispensable calculations (see also Section 6.5.1).

6.3.5 Open Problems

Even though the *TimeScale* is fast and is able to provide a correct visual representation for all possible spans, there is still one problem left. When a *view coordinate* is transformed to a *screen coordinate* it is automatically rounded to integer positions (pixels). *View coordinates* are defined in double precision, that means that each coordinate has an associated *absolute coordinate* and therefore an associated date. In contrast, a *screen coordinate* is an interval of *view coordinates* as countless of *view coordinates* can be rounded to the same *screen* point.

The time-cursor has only access to its actual *screen* position. When this cursor requests its associated date from the *TimeScale* the exact date of the integer is returned. But, the returned date is not related to the actual granularity (e. g. , even though the actual granularity is “years”, the returned date is accurate to milliseconds). This behavior is not wished, nevertheless, a solution to align associate dates of screen points was not found yet.

6.4 Visual Structures

Once, the underlying tree structure contains at least one node, visualization can start. As in the data Section, the *ItemRegistry* plays an important role within this process. Its responsibility is to store and provide *VisualItems* in a separated tree.

VisualItem is the *Prefuse* top level class for all visual elements, *NodeItems* and *EdgeItems*. A *Display* is the view of an *Information Visualization* done with *Prefuse*. It is able to represent *VisualItems*.

To create a view, there are two different data transformations necessary, filtering and rendering. Filtering is the process of deciding which (abstract) nodes and edges are visualized and creating visual analogues of them. This process is initialized on users’ interaction when expanding or collapsing already existing nodes. The result of the filtering process is a second tree stored in the *ItemRegistry*, containing all visual structures that may be rendered.

In contrast to filtering, rendering is initialized by the *Display* whenever a repaint of the view is necessary. The *Display* is an extended *JComponent* and therefore a Java AWT element. Repainting of this component is forced on interactions like resizing, change of visibility, etc. But also the window-manager may force a repaint on mouse or keyboard events. Whenever the component has to be repainted, also the rendering of available *VisualItems* is done to provide an updated view.

6.4.1 Filtering the Tree

Filtering is the process of retrieving or creating visual analogues of abstract nodes and edges when navigating in the hierarchy (see Section 4.2.3). As mentioned, this process is only performed on collapsing or expanding an already existing visual node on the *Display*. This happens on a click on the expanding-symbol of a *PlanningLine* or on double-click on it (see *Control-Listeners* 6.4.4).

Filtering is done by performing two different actions: the filtering itself and a following layouting of all visual structures. Actually, there is a third one, the repainting, but this action is not of special interest as it just activates the re-rendering of the view (see Section 6.4.3).

These three actions are combined in an *ActionList*. An *ActionList* is a runnable container that executes defined *Actions* in a specified order.

In case of expanding a node, following tasks are performed:

1. If children are not loaded yet, retrieve abstract data of the *AbstractDataTable*, create respective *PlanningTreeNode*s and *PlanningTreeEdge*s, and insert them within the *PlanningTree*. At this time, no visual analogues of them exist.
2. Change visible state of already existing visual children of the node. The visible state of children may be set to false as they were collapsed earlier (remember, as the visual structures are already created, thus they were also visible before).
3. Call the filter *ActionList* to create *VisualItems* of newly created nodes and edges and perform a new layout for all visible items.

Collapsing nodes is done the same way, except that it is not necessary to create new visual items.

PlanningTreeFilter

Filtering is done by the *PlanningTreeFilter* class which is defined as *Action*. The filter forces the *ItemRegistry* to provide (and if necessary create) visual structures for all abstract nodes and edges stored within the *PlanningTree*. Once a *PlanningTreeVisualNode* or *PlanningTreeVisualEdge* is created, it stays in memory of the *ItemRegistry*.

Of course it would be possible to hold only visible items within the *ItemRegistry*, but creating of visual structures involves a lot of calculations, and, the possibility of expanding/collapsing nodes causes changes of other visual items, even if they are not visible. Even though memory may be wasted, it seems that a permanent storage of visual items is the only way to guarantee a high level of performance and meet the requirements of the polythetic design.

The filter creates a second tree (of the type *DefaultTree* as no special functionality is needed) containing all *PlanningTreeVisualNodes* as well as *PlanningTreeVisualEdges* and writes this tree back to the *ItemRegistry* as filtered graph. Besides the creation of new *VisualItems*, the filter may also change the visible states of edges:

- **EdgeItem:** As these edges just map the hierarchical structure, all visual representations of them are set to invisible.
- **PlanningTreeVisualEdge:** These edges represents logical relationships between tasks. If both nodes, predecessor and successor, are visible, the state is set to visible, otherwise theses edges are invisible too.

PlanningTreeLayouter

The second *Action* that is executed while filtering is the *PlanningTreeLayout*. This *Action* just has to maintain the vertical alignment of *VisualNodes*, as the horizontal alignment is given by the *TimeScale*.

Setting the vertical alignment is done by an algorithm considering the visible state of a node, hierarchical decomposition, logical relations, and also time facets like start or duration. Note that the *PlanningTreeVisualEdges* do not have to be layouted, as their position depends on the position of the nodes a edge connects.

The algorithm to layout *PlanningTreeVisualNodes* follows this rules (in the given priority):

1. If an item is not visible, the item and all its children are ignored in the calculation.
2. Align children right below the respective parent.
3. The ordering of the children requires the consideration of logical relations. Items that are connected with another item should lie next to each other. If this is not possible because of other logical relations, the space between them must be minimized.
4. The earlier an item starts, the nearer it should be placed next to its parent.

It is possible that edges cross nodes, but the *PlanningTreeLayout* tries to prevent the unwanted crossings.

Subsequently, the whole view is repainted. This is also done by starting the *RepaintAction*.

6.4.2 Visual Structures in ItemRegistry

As mentioned, the filter creates a tree containing all visual structures. This tree is stored within the *ItemRegistry*, so *Actions* and *Renderers* can access it. Like the abstract structures, all visual structures implement the *Node* or *Edge* interface. These interfaces provide all functionality to hold implementations of them within a graph structure of *Prefuse*. Furthermore, all structures are inherited from the class *VisualItem*. This top level class provides the typical functionality of a visual structure, like assignment attributes or painting issues.

PlanningTreeVisualNodes and PlanningTreeVisualEdges

The *PlanningTreeVisualNode* is the visual analogue to the *PlanningTreeNode* and implements the *PaintingObject* interface (even though the paint method is not implemented, as this is done by a *Renderer*). It is initialized with its associated *PlanningTreeNode* that provides all necessary abstract data that is needed to create a set of visual attributes. Besides common attributes like visible state, all assignment attributes are of special interest:

- **X:** The x-coordinate depends on the Earliest Starting Time (EST) of the task. During initialization, the *TimeScale* is asked for it.
- **Y:** The y-coordinate is set to 0 as it is later calculated by the *Layouter*. As the vertical alignment depends on the expanding/collapsing state of an item's parent, this coordinate can change whenever the graph is filtered.
- **Width:** Like the x-coordinate, width also depends on the *TimeScale* and is stored as deviation to the x-coordinate.
- **Height:** The height is a fixed value in pixels and the same for all items.

It is important to note that all coordinates except the y-coordinate are calculated once, with regards to the *absolute coordinate system*. Also zooming or panning does not affect these coordinates, since all transformations are performed using the affine transformation matrix of the graphic context, therefore only *view coordinates* are affected.

But, as a *PlanningLine* is more complex than a simple rectangle, a *PlanningTreeVisualNode* is composed of different visual elements implementing the *PaintingObject* interface. Therefore, initializing the node itself also means that *absolute coordinates* of caps, duration bars, the expanding-symbol, etc., must be calculated and stored in the respective visual element.

Besides assignment attributes, a visual element also provides a paint method used for rendering. As at least the y-coordinate of the *PlanningTreeVisualNode* can change, these changes must be passed to all visual elements a *PlanningTreeVisualNode* is composed of.

This is obtained by a layered structure of single visual elements, where all positions and dimensions are given in relation to the coordinates of the topmost element, the *PlanningTreeVisualNode*. Concrete, these visual “children” are:

- **NotRendered:** A visual element that is rendered instead of the *PlanningLine* when the *PlanningLine* is too small. It would not make sense to render a *PlanningLine* which is so small that no details are recognizable. The minimum width set in the prototype is 30 pixels.
- **LifeLine:** A rectangle filling the whole task. Generally, this element is rendered using the background color of the view so it is not visible. But there is a second form of the *LifeLine*, it can also appear as summary bar to indicate hierarchical decomposition.
- **IntervalCap:** Each node has two *IntervalCaps* representing the start and end-interval of a task. The *IntervalCap* can also appear as small diamonds, indicating that the interval is not given completely.
- **ExpandingSymbol:** A small triangle indicating that the *PlanningLine* can be expanded to make children visible.
- **LifeLineLabel:** A label displaying the name of a task.
- **DurationBar:** Each node consists of two *DurationBars*, one for the minimum duration, the other one for the maximum duration.

Typically, a *PaintingObject* is initialized with a reference to another *PaintingObject*, called parent. Positions of an element are specified relative to the position of the parent. The layered structure regarding to assignment attributes means (see also Figure 6.8):

- *NotRendered*, *IntervalCaps*, and the *LifeLine* refer to the *PlanningTreeVisualNode*.
- *ExpandingSymbol*, *LifeLineLabel*, and the maximum *DurationBar* refer to the *LifeLine*.
- The minimum *DurationBar* refers to the maximum *DurationBar*.

A *PlanningTreeVisualEdge* is implemented in a similar way. It is also a *PaintingObject* and holds the visual element *ConnectArrow*. Assignment attributes are calculated regarding to its predecessor and successor, also

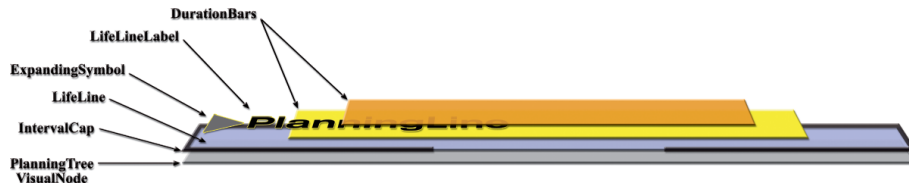


Figure 6.8: Layer structure of a PlanningTreeVisualItem.

the *ConnectArrow* refers to these objects. This ensures that coordinates of arrows representing logical relations always go conform with the connected *PlanningTree VisualNodes*. Therefore, *PlanningTree VisualEdges* must not be considered when laying out the view.

Figure 6.9 gives an overview about the implementation of visual structures using *Prefuse* and the interface *PaintingObject*.

6.4.3 Rendering of Visual Structures

Rendering is always necessary when the view of the *Information Visualization* must be repainted. Each *VisualItem* has an associated *Renderer*, in this case the *PlanningTreeNodeRenderer* and the *PlanningTreeEdgeRenderer*.

Renderers are maintained by the *RendererFactory* which is maintained by the *ItemRegistry*. On creation of a *VisualItem*, the *ItemRegistry* assigns the correlating *Renderer* to the item (see also Figure 6.10).

In case of repainting, the paint method of the *PlanningLinesDisplay* is called. As the *Display* is associated with an *ItemRegistry*, it asks for all visible *VisualItems* (both, nodes and edges) stored in the filtered tree. In the following, the associated *Renderer* of each item is accessed. A *Renderer* typically has two important tasks:

1. Providing a bounding box that defines the location and maximal size an item needs when painted.
2. Providing functionality for painting the actual area using a Java *Graphics2D* object.

The bounding box is used by the *Display* to decide which items must be rendered and which ones can be discarded as they would not be visible anyway. This way of delimitation the count of items to be painted is often called bounding-management. In case of a *PlanningTree VisualNode* the area is given by the position and dimension of the node, in case of the *PlanningTree VisualEdge* the area is the space between the two nodes connected by the edge.

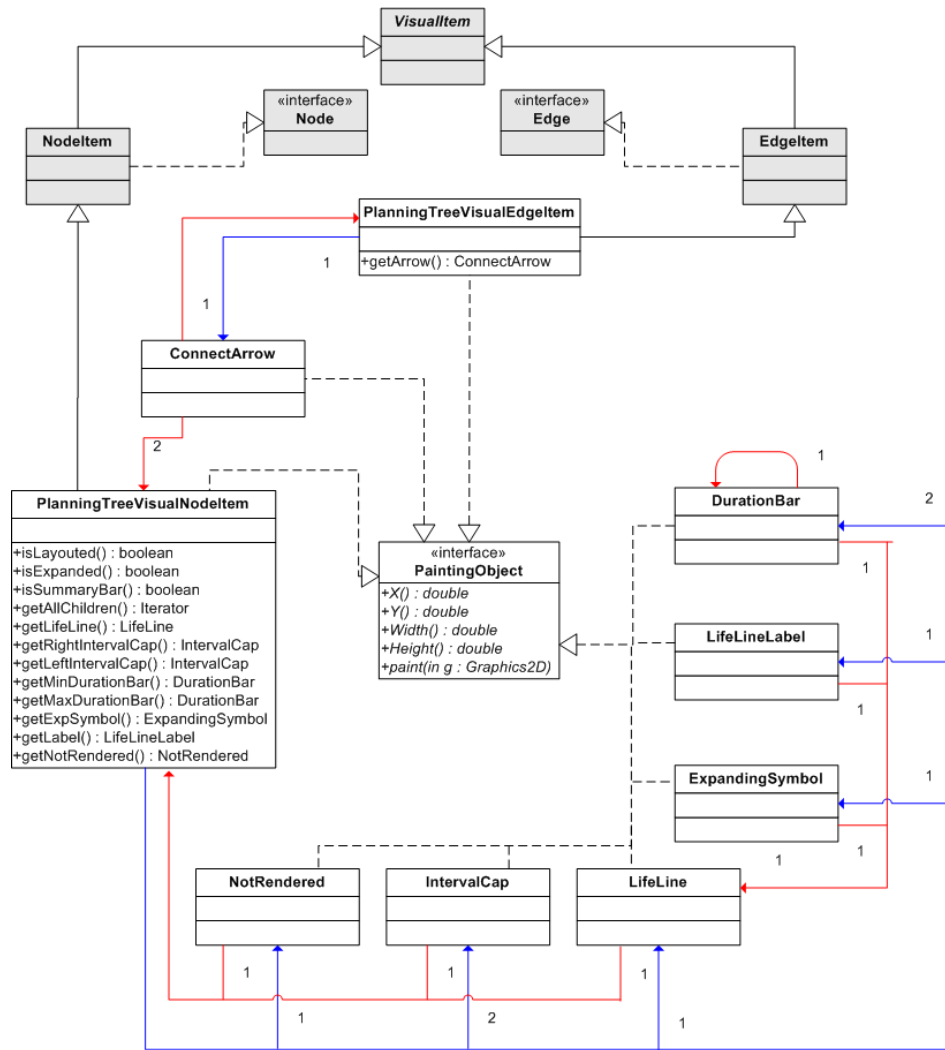


Figure 6.9: UML diagram of nodes and edges. Blue cardinalities represent the visual elements held by a **VisualItem** including the exact count. A red cardinality illustrates the object the actual visual element refers to.

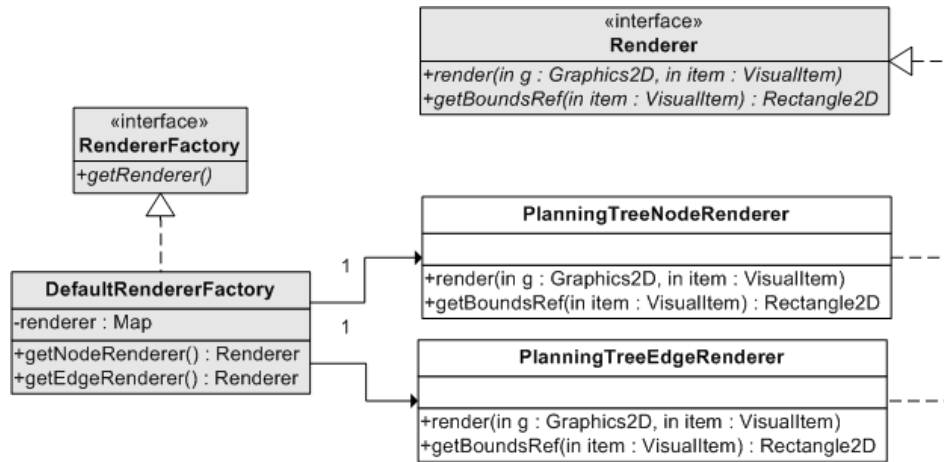


Figure 6.10: UML diagram of the Renderers.

If the bounding box of an *VisualItem* intersects the visible area of the *PlanningLinesDisplay*, the respective *Renderer* is called again to render the visual structure.

Painting is done using a Java *Graphics2D* object. The *Display* passes its own *Graphics2D* object to the different *Renderers*. Depending on the selected view (see also Section 5.2), the *Renderer* decides which of the *Painting-Objects* a *VisualItem* is composed of must be painted. As each *Painting-Object* is able to paint itself, the *Renderer* just has to call the paint methods of the selected elements in the right order.

6.4.4 View

The view of a *Prefuse Information Visualization* is created using the *Display*, an extended class of the Java Swing *JComponent*. The prototype itself extended this class again to make some adjustments (*PlanningLinesDisplay*).

A *Display* is initialized with the *ItemRegistry* it refers to. The *PlanningLinesDisplay* has to maintain following tasks:

1. Rendering the *Information Visualization*. This includes painting the background, the items, and an additional foreground (in this case a time-cursor).
2. Maintain a set of *ControlListeners* to handle user interactions.
3. Provide functionality of navigation. This is done with an affine transformation matrix of the *Display*. Therefore, also functionality for modifying this matrix must be provided.

As the top level class *JComponent* provides a lot of user interface functionality, this is not outlined here. Only the graphical issues and event handling of the *Display* are really important to understand. Several events can cause rendering or repainting the view:

- Standard Java AWT repainting, caused on events like resizing or getting the focus.
- An explicit call of the repaint method by an underlying layer or by *ControlListeners* after they have finished their work and the results must be communicated to the user.
- On a *RepaintAction* that forces the *ItemRegistry* to repaint all related *Displays*.

Basically, the default *Display* of *Prefuse* provides most of the needed functionality. But the demand of a time-cursor and a background extending the ticks of the *TimeScale* forced an extension of the original paint methods, realized with the *PlanningLinesDisplay* (see Figure 6.11)

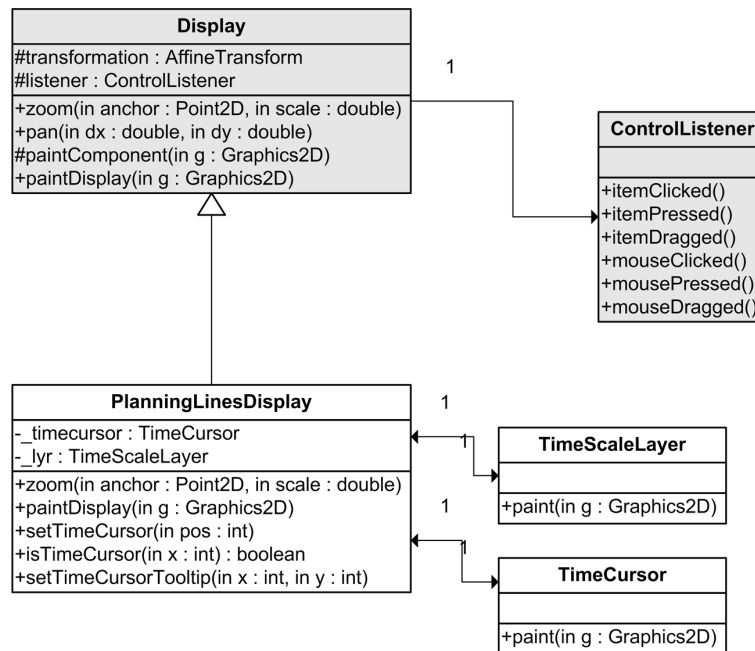


Figure 6.11: UML diagram of the *PlanningLinesDisplay*.

Rendering

The rendering of items like nodes or edges is described in the Section above. But, before performing this task, the background has to be painted. The

background of the *Display* continues the tick-lines of the *TimeScale*, helping users to identify temporal aspects of overlaying items.

Therefore, the *PlanningLinesDisplay* has to retrieve the positions of the *TimeScale* to fill its own background with lines. This routine is only performed after the *TimeScale* has changed. In the meantime, the lines are held in an own offscreen picture to avoid useless painting and calculations (see *Double Buffering*, Section 6.5.1).

After painting the background, the entire rendering of *VisualItems* is performed. Subsequently, a single red line, called time-cursor, is painted on the top of the view. The *TimeCursor* class holds an offscreen picture containing a single line. The painting method just has to paint this picture on the graphics context of the *Display* and an additional label for the actual date represented by the *TimeCursor*. The *TimeCursor* is completely independent of zooming or panning, therefore its position does not change when a user navigates.

The standard Java *Double Buffering* of the *JComponent* is disabled, instead of *Double Buffering* is managed by the *Display* itself to provide maximal possible performance of painting jobs.

ControlListeners

User interactions are realized by adding some *ControlListeners* the *Display*. A *ControlListener* is an object which can handle one or more keyboard and mouse events. Whenever the *Display* receives such an event, it passes it to the respective *ControlListener*.

Basically, there are two kinds of events that are distinguished: general events that affect the whole view and events that concern a certain item. The application provides five different *ControlListeners*:

- **ToolTipControl:** This control is provided by *Prefuse* and maintains tooltips for *VisualItems*.
- **CursorControl:** This listener is responsible for displaying the adequate mouse cursor depending on mouse position and additional keys that are pressed by a user. Additionally, this *ControlListener* also cares about the tooltip of the *TimeCursor* as this is not a visual element of *Prefuse*.
- **ClickControl:** Treats all mouse-clicks on the view. This *ControlListener* decides when a hierarchical decomposed node has to be collapsed or expanded.
- **DoubleClickControl:** This *ControlListeners* handles double-clicks on *PlanningTreeVisualNodes*. A double-click is another possibility to expand or collapse a node.

- **DragControl:** Handles all dragging events of the view. The action depends on the mouse-button clicked while dragging. If it is the left button, panning is performed. If it is the right one the zoom factor of the view changes. The control is also responsible for moving the *Time-Cursor*. An additional implemented feature is the moving of duration bars within their bounds.

Figure 6.12 illustrates the functionality of *ControlListeners*. All listeners are maintained by the *Display*. This is done by a *MultiCaster*, which merges all defined *ControlListeners* (see also Figure 6.11).

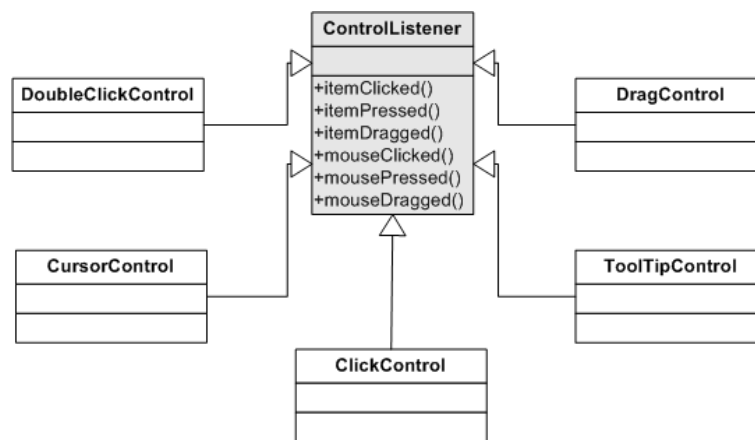


Figure 6.12: UML diagram of the applied *ControlListeners*.

Typically, each event is ended by calling the repaint method of the *Display* to make changes visible.

Navigation

Generally, navigation within the view is done on user interactions (thus by a *ControlListener*). Available navigation techniques are panning, zooming, and navigating within the hierarchy. The *Display* provides several ways of applying one of these techniques to the view. It is possible to specify an exact point the view should pan to, but it is also possible to apply delta values. Both techniques can be used animated too (by defining a duration the process should last).

The *PlanningLinesDisplay* does not use standard zoom methods provided of *Prefuse*. Instead, an own method is applied that disables vertical zooming. When zooming is applied in both directions, on a high zoom factor the whole visual representations would only consist of a small part of a single *PlanningLine* that is stretched. As this would be useless, a *PlanningLine* is only stretched horizontal, its (visual) height keeps constant.

Technically, both techniques just make adjustments to the underlying affine transformation matrix of the *Display*. The matrix has several values that can be changed to modify the *view coordinate system* and therefore the whole view communicated to users (see Section 6.5.2).

This application does not work with animated navigation. Instead, the *DragControl*, which is responsible for these issues, provides a smooth way of panning and zooming. This is reached by using delta factors that depend on mouse movements to change the entire view. This allows a user to decide how fast or slow navigation is done, therefore a better feeling for these techniques is ensured.

The *ClickControl* and *DoubleClickControl* just listen for interactions that change the expand/collapse state of an item. Whenever this event occurs, the filtering process is executed.

6.5 Graphical Concepts

Concluding the chapter about the architecture, some graphical methods used within this prototype are outlined. As these are basic concepts used in visualization, nearly all visual classes deal with these concepts.

6.5.1 Double Buffering

Double Buffering is a standard concept applied in paint methods of Java components. Its goal is to avoid flickering of the area that is painted. Also the performance of visualization may improve, as often needed graphical elements can be stored and must not be repainted whenever needed.

In Java each graphical element has a graphic context (e. g. , a *Graphics2D* object). This context can belong to the element itself or it is passed by another graphical element. Typically, painting executes two steps:

1. Erasing the background of the area which must be repainted. That means, previous paintings (that are still displayed) are replaced with a background color.
2. Doing the new painting. In fact, the previously deleted area has to be filled with single pixels.

A computer screen redraws itself about 60 to 100 times a second. Therefore, it is hard to execute the painting job before the next repaint or the screen is performed. This leads to incomplete graphics a user can recognize. Especially, the erasing of the background causes flickering, as the whole part of a view is exchanged with a single-colored area.

The performance of today's computers is capable of performing very simple graphics in a time, the flickering effect is not visible that much. But

more complex graphics, and especially graphical animations like panning cause an unacceptable flickering of a graphical user interface [Sun Microsystems, 2001].

The idea of *Double Buffering* is to perform all painting jobs in the background. This is obtained by a *BufferedImage* which size is identical to the area that has to be filled with graphical elements. This picture is often called offscreen. All painting jobs are done on this image, while the screen area stays as it is. Once, the image painting has finished, the complete image is displayed at once on the screen.

Drawing a whole image is a very fast process. The flickering disappears nearly completely, as the old area stays unchanged on the screen until all calculations of the image have finished and the area is exchanged at once.

In complex visualizations, the whole screen is created using a lot of single offscreen pictures (e.g., composing a visual structure with a lot of offscreen pictures, the structure itself is an offscreen picture again, the view). Such a composing of a (layered) view is supported by the ability of transparent backgrounds. This is the only way to provide a fast rendering of a large amount of complex visual structures.

6.5.2 Affine Transformation

The concept of affine transformation is widely used in computer visualization, especially when mapping a 2D object to an image [Bebis et al., 1999]. This is actually done by the painting of objects.

As described in Section 6.1.2, there are two different coordinate systems applied in this visualization. The *absolute coordinate system* is used for placing and dimension issues, it is also called user space [Sun Microsystems, 2001]. All elements of this application are initialized regarding to this system.

In contrast to the user space, the device space is used for placing elements visually on the screen. The transformations from *absolute* to *view coordinates* are done by the graphics context automatically. Therefore, the graphics context uses a matrix describing the deviation of *view coordinates* to *absolute* ones. This is realized with an object called *AffineTransform*.

When painting, the graphics context receives *absolute* assignment information like coordinates or dimensions. These *absolute* values are passed to the matrix which returns the associated *view* value.

Mapping between *absolute* and *view coordinates* is done linear. An affine transformation changes all values using the same formulas. Parallel lines stay parallel, relative proportions are retained [Austin, 2003].

Furthermore, *view coordinates* are quoted in the following way: x' and y' . Generally, the *view coordinate* system is described using following factors for both dimensions:

- *Scale*: This factor is a multiplier used for scaling *view coordinates* (*sx* and *sy*). This factor is responsible for zooming.
- *Translate*: Describes a translation in a certain direction (*tx* and *ty*). This factor is responsible for panning.
- *Shear*: Describes the shearing points (*shx* and *shy*). This, in combination with scaling, is used for rotations.

Mathematically, this is solved using a 3x3 matrix. Transformations are calculated using the following model (the last row is a dummy row needed to perform matrix calculations):

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} sx & shx & tx \\ shy & sy & ty \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

This matrix is equivalent to the following two statements that show how *view coordinates* are calculated [Sun Microsystems, 2001]:

$$x' = sx * x + shx * y + tx$$

$$y' = shy * x + sy * y + ty$$

Considering the functionality of the prototype that means that panning changes the translation factors and zooming the scale factors. As zooming is only provided in the horizontal direction, *sy* is equal to 1. This causes some problems, as the automatic painting routines of Java are not able to determine how an item should be represented if zooming is not applied in both directions. This concerns elements like vertical lines that get too wide and labels that are not readable anymore as the proportions are not right (stretched in the vertical direction).

To prevent this behavior, some of the elements must be painted without using the transformation matrix. But this also means that the *absolute coordinates* of an element must be converted manually to *view coordinates*.

Whenever such an element has to be painted, a blank transformation matrix is set to the *Graphics2D* element before painting on it. Thus, no coordinates passed to the graphics context are transformed anymore. Therefore, lines, etc., are painted in a normal way, as neither zooming nor panning is applied to the graphics context.

As automatic routines are bypassed by setting a blank matrix, the *view coordinates* have to be calculated manually by the element. The *Coordinate* class provides this functionality. The received *view coordinates* are passed to the *Graphics2D* object and the visual element is drawn. As the graphics context does not perform any transformations again, the element is painted the same way as there would be no zooming applied to the view.

Finally, the original transformation matrix is set back to the *Graphics2D* element, so further paintings can perform their transformations automatically.

With describing the architecture of the prototype the end of this thesis is reached. The next Chapter recapitulates the work that has done.

Chapter 7

Conclusion

7.1 Summary

Temporal uncertainties are present in every plan as the future cannot be predetermined exactly. There are several ways how to consider such uncertainties, but, plans may become very complex when uncertainties flow into them (especially large plans). Even though PERT considers temporal uncertainties and provides methods to analyse them, typically, only slack times between tasks are considered in practice. The missing graphical notation of these uncertainties makes it hard to work with a PERT chart that contains the full range of possible temporal facts.

The implemented prototype tries to fill this gap of treating temporal uncertainties visually. The concept *PlanningLines* which provides a graphical notation of temporal uncertainties was applied in a modern and sophisticated *Information Visualization*. An *Information Visualization* amplifies the cognition and perception of complex information [Card et al., 1999]. In case of temporal uncertainties within a plan, the task of an *Information Visualization* is to communicate these complex coherencies to users so they are able to understand them and see possible problems at a glance.

The prototype is able to display *MS-Project* or *Asbru* plans. Applied interaction techniques allow users to navigate within a plan. All facts are displayed graphically, therefore, the visual sense and the ability of conceiving logical patterns are used directly to communicate complex plans.

Besides displaying a plan by using *PlanningLines*, also *LifeLines* can be chosen as notation. Furthermore, the prototype supports two different modes to display hierarchical decomposition.

Recapitulating, it seems that *PlanningLines* is a proper way of treating temporal uncertainties visually. The implemented prototype combines this concept with methods and techniques of *Information Visualization*. This is a first attempt to evaluate this concept in a modern visualization. Further, ways of technical realization of such a software were shown.

7.2 Evolution

The main goal of this thesis was the development of a prototype which applies the concept *PlanningLines* to *MS-Project* and *Asbru* plans. PERT is the leading charting technique to treat plans with temporal uncertainties. However, *PlanningLines* is a concept that tries to treat such uncertainties graphically instead of textually as PERT does.

Techniques and methods of *Information Visualization* were of special interest as they directly flow into the development of the prototype. Especially, the usage of the *InfoVis Pipeline* was a central aspect when developing the prototype [Card et al., 1999]. The *InfoVis Pipeline* affected the whole implementation, starting with the selection of adequate toolkits and ending in the whole architecture which is conform to this recommendation. Firstly, it seemed that the *InfoVis Pipeline* provides some additional work, but once some details were worked out the consideration of the *InfoVis Pipeline* provided a lot of advantages with regards to extensibility. Additionally, the *InfoVis Pipeline* ensured a well designed application which may be used in further development.

The selection of adequate toolkits was an important task. After some attempts with *Piccolo*, *Prefuse* was chosen to be the main framework for implementing the *Information Visualization*. *Prefuse* provides most of the wanted functionality a priori to support the development of the prototype. Quickly a way was found to create *PlanningLines*, also assignment of them on the view was easy to implement. Nevertheless, some problems occurred while development.

In the beginning, zooming was applied in both directions. This is a common technique in *Information Visualization* to support navigation in a view. In case of *PlanningLines* it turned out that zooming in the vertical direction makes no sense as single *PlanningLines* get too large. Furthermore, a loss of provided information is caused since coherencies between *PlanningLines* may not be conceivable anymore because of their size. The decision of disabling vertical zooming caused some problems with the automatic rendering routines of Java. Therefore, a lot of work was investigated to solve these problems. But, in the end this additional work was rewarded as surrounding areas can still be spotted even if a certain task or area is displayed in detail.

Another challenge was the development of the timescale. Firstly, it seemed that this would be an easy work. But the fact that dates are not linear caused some problems in the graphical representation. Therefore, ways had to be found to consider calendar anomalies without losing performance in painting issues. However, there is still a problem left, the providing of an associated date to a certain screen point.

After most of these problems were solved, the prototype displays plans with temporal uncertainties visually. So problems or possibilities to lower the whole duration of a plan can be seen at a glance. Nevertheless, by the

use of tooltips and navigation techniques (zooming and panning) also details of the plan can be extracted easily.

7.3 Learned Lessons

Visual representation of complex data is often the only possibility to conceive the coherencies. The discipline *Information Visualization* provides a lot of concepts and methods to fulfill this task. Further, developers of an *Information Visualization* are supported by several free or commercial products that can facilitate the implementation.

Nevertheless, developing of an *Information Visualization* is hard work. A lot of different things have to be considered to result a usable application. This work has shown that concepts of *Information Visualization* (like the *InfoVis Pipeline*) or the usage of complex toolkits are reasonable even if some additional work has to be performed when implementing the basis. But, at least when some details must be solved this additional work is rewarded.

Chapter 8

Future Work

Since there is a new release of *Prefuse* available, a porting to the new framework would provide several advantages. On the one hand, this version supports new functionality, on the other hand, the design of it is more lightweight and flexible. Furthermore, the new framework considers the *InfoVis Pipeline* even more than the old one as data tables are supported as basic data structure. Therefore, when the prototype is ported, the *Prefuse* table could replace the *InfoVis Toolkit* table which would make the whole architecture more reasonable.

Especially, the timescale demands some enhancements since the current behavior of providing associate dates to screen points is not satisfying. A way has to be found where the actual granularity is considered, but, this will be a real challenge.

Some missing functionality was already mentioned in Section 5.3. Besides applying more flexibility (e. g. , changing of visual attributes at runtime, an overview window, etc.), the integration of the prototype in *CareVis* is the most important issue.

Once the functionality is enhanced, user testings should be performed. An evaluation study with domain experts would reveal how the design and navigation is accepted and where improvements are necessary.

Appendix A

Indeterminacies Calculation Table

EST	LST	EFT	LFT	minDu	maxDu	Applied Formulas
X	X	X	X	X		(3.3)
X	X	X	X		X	(3.1), (3.2)
X	X	X		X	X	(3.9)
X	X		X	X	X	(3.5)
X		X	X	X	X	(3.4)
	X	X	X	X	X	(3.8)
X	X	X	X			(3.1), (3.2), (3.3)
X	X	X			X	(3.1), (3.2), (3.9)
X	X			X	X	(3.5), (3.9)
X			X	X	X	(3.6)
		X	X	X	X	(3.4), (3.8)
X	X		X	X		(3.3), (3.5)
X		X	X	X		(3.3), (3.4)
	X		X	X	X	(3.5), (3.8)
	X	X	X		X	(3.1), (3.2), (3.8)
X		X		X	X	(3.4),(3.9)

Table A.1: The Table illustrates which formulas are applied when calculating indeterminacies or a PlanningLine. The equation numbers refer to the formulas given in Section 3.3.1. Note, that Equation 3.2 is only performed when PlanningLine has overlapping intervals. The Table is continued on the next site.

EST	LST	EFT	LFT	minDu	maxDu	Applied Formulas
X	X	X				(3.1), (3.2)
X	X				X	(3.9)
X				X	X	(3.6), (3.9)
			X	X	X	(3.7), (3.8)
X	X		X			(3.3)
X		X	X			(3.3)
X	X			X		(3.5)
	X	X	X			(3.1), (3.2)
X			X	X		(3.3), (3.6)
	X		X	X		(3.5)
X		X		X		(3.4)
		X	X	X		(3.4)
X		X			X	(3.9)
	X	X			X	(3.1), (3.2)
	X		X		X	(3.8)
		X	X		X	(3.8)
	X			X	X	(3.5)
		X		X	X	(3.4)
	X	X				(3.1), (3.2)
X			X			(3.3)
X				X		(3.6)
	X			X		(3.5)
		X		X		(3.4)
			X	X		(3.7)
X					X	(3.9)
			X		X	(3.8)

Table A.2: Continuation of table A.1;

Appendix B

UML Notation

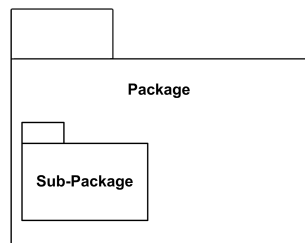


Figure B.1: A package which contains a sub-package.

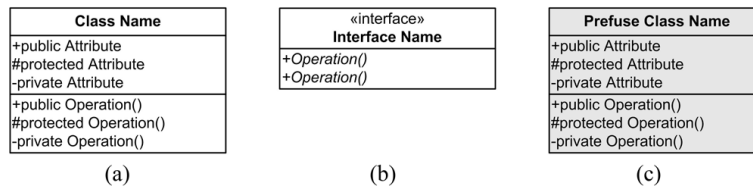


Figure B.2: (a) shows a class, (b) an interface, (c) a class of prefuse.

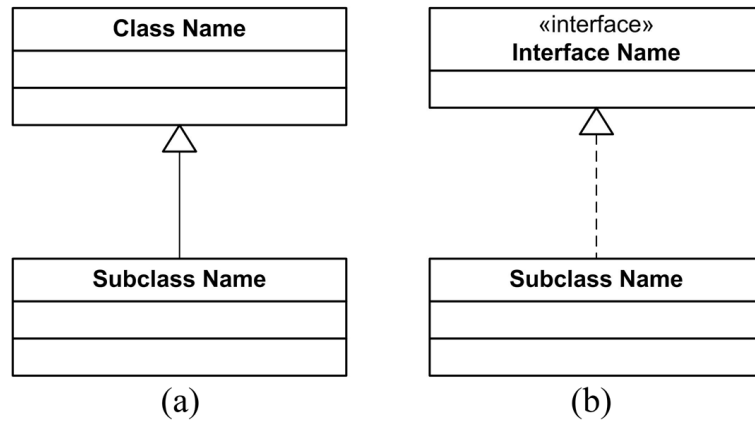


Figure B.3: (a) shows an extended subclass, (b) a subclass that implements an interface.

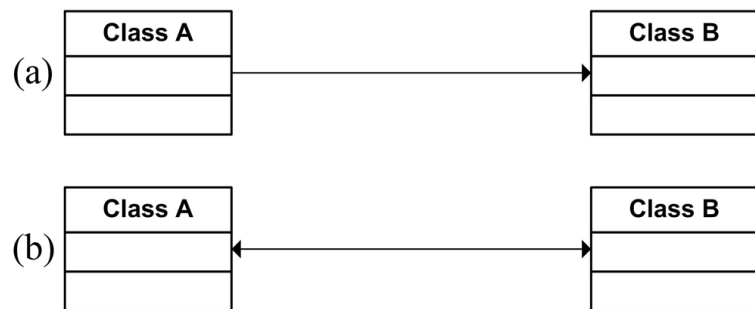


Figure B.4: (a) shows an association in one direction (Class A holds a reference to Class B), (b) shows an association in both directions (Class A holds a reference to Class B and vice versa). Further, associations may depict cardinality (e. g. , 0..1, 1, *).

Bibliography

- [Aigner, 2003] Aigner, W. (2003). Interactive Visualization of Time-Oriented Treatment Plans and Patient Data. Master’s thesis, Vienna University of Technology, Austria.
- [Aigner and Miksch, 2004] Aigner, W. and Miksch, S. (2004). Communicating the Logic of a Treatment Plan Formulated in Asbru to Domain Experts. In *Computer-based Support for Clinical Guidelines and Protocols (CGP 2004)*, pages 1–15.
- [Aigner and Miksch, 2006] Aigner, W. and Miksch, S. (2006). CareVis: Integrated visualization of computerized protocols and temporal patient data. *Artificial Intelligence in Medicine*, 37(3):203–218.
- [Aigner et al., 2005a] Aigner, W., Miksch, S., Thurnher, B., and Biffl, S. (2005a). PlanningLines: Novel Glyphs for Representing Temporal Uncertainties and Their Evaluation. In *9th International Conference on Information Visualisation, IV 2005, 6-8 July 2005, London, UK*, pages 457–463.
- [Aigner et al., 2005b] Aigner, W., Miksch, S., Thurnher, B., and Biffl, S. (2005b). PlanningLines Usability Studie - User Study zum Vergleich von PlanningLines und PERT Darstellung (in German). Technical Report Asgaard-TR-2005-3, Vienna University of Technology.
- [Austin, 2003] Austin, D. (2003). Graphics2D’s Internal State - Lecture Notes. <http://merganser.math.gvsu.edu/david/reed03/notes/chap4.pdf>. Online; accessed April 20th, 2006.
- [Bebis et al., 1999] Bebis, G., Georgiopoulos, M., da Vitoria Lobo, N., and Shah, M. (1999). Learning affine transformations. *Pattern Recognition*, 32(10):1783–1799.
- [Card et al., 1999] Card, S. K., Mackinlay, J. D., and Shneiderman, B. (1999). *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.

- [Chi, 2000] Chi, E. H. (2000). A Taxonomy of Visualization Techniques Using the Data State Reference Model. In *INFOVIS*, pages 69–76.
- [Duftschmid, 1999] Duftschmid, G. (1999). *Knowledge-Based Verification of Clinical Guidelines by Detection of Anomalies*. PhD thesis, Vienna University of Technology, Vienna.
- [Fekete, 2004] Fekete, J.-D. (2004). The InfoVis Toolkit. In *10th IEEE Symposium on Information Visualization (InfoVis 2004)*, pages 167–174.
- [Fekete, 2006a] Fekete, J.-D. (2006a). The InfoVis Toolkit Homepage. <http://ivtk.sourceforge.net/>. Online; accessed April 16th, 2006.
- [Fekete, 2006b] Fekete, J.-D. (2006b). The InfoVis Toolkit (Rapport de recherche, INRIA Futures). <http://www.inria.fr/rrrt/rr-4818.html>. Online; accessed April 16th, 2006.
- [Friedland and Iwasaki, 1985] Friedland, P. and Iwasaki, Y. (1985). The Concept and Implementation of Skeletal Plans. *J. Autom. Reasoning*, 1(2):161–208.
- [Gareis, 2000] Gareis, R. (2000). Managing the Project Start. In *The Gower Handbook of Project Management*, pages 451–467. Aldershot.
- [Hadorn, 1995] Hadorn, D. (1995). Use of Algorithms in Clinical Guideline Development in Clinical Practice. In *Guideline Development: Methodology Perspectives (AHCPR Pub. No. 95-0009)*, pages 93–104.
- [Harrington, 1991] Harrington, H. J. (1991). *Business Process Improvement: The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*. McGraw Hill, 1. edition.
- [Heer, 2004] Heer, J. (2004). Prefuse: A Software Framework for Interactive Information Visualization. Master’s thesis, University of California, Berkeley, USA.
- [Heer et al., 2005] Heer, J., Card, S. K., and Landay, J. A. (2005). Prefuse: A Toolkit for Interactive Information Visualization. In *Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005*, pages 421–430.
- [Joint Publications, 2001] Joint Publications (2001). Department of Defense Dictionary of Military and Associated Terms. http://www.fas.org/irp/doddir/dod/jp1_02.pdf. Online; accessed March 18th, 2006.
- [Keim and Kriegel, 1996] Keim, D. A. and Kriegel, H.-P. (1996). Visualization Techniques for Mining Large Databases: A Comparison. *Transactions on Knowledge and Data Engineering, Special Issue on Data Mining*, 8(6):923–938.

- [Kosara et al., 2001] Kosara, R., Messner, P., and Miksch, S. (2001). Time and Tide Wait for No Diagram. Technical Report Asgaard-TR-2001-2, Vienna University of Technology.
- [Kosara and Miksch, 2001] Kosara, R. and Miksch, S. (2001). Metaphors of Movement: A Visualization and User Interface for Time-oriented, Skeletal Plans. *Artificial Intelligence in Medicine*, 22(2):111–131.
- [Messner, 2000] Messner, P. (2000). Time Shapes - A Visualization for Temporal Uncertainty in Planning. Master's thesis, Vienna University of Technology, Austria.
- [Meyer, 1997] Meyer, W. (1997). Information Visualization Prinzipien und Techniken (in German). Master's thesis, Vienna University of Technology, Austria.
- [Microsoft, 2006] Microsoft (2006). Microsoft Office Online. <http://office.microsoft.com/en-us/default.aspx>. Online; accessed March 15th, 2006.
- [Miksch, 1999] Miksch, S. (1999). Plan management in the Medical Domain. *AI Commun.*, 12(4):209–235.
- [Modell, 1996] Modell, M. E. (1996). *A Professional's Guide to Systems Analysis*. McGraw-Hill, Inc., Hightstown, NJ, USA, 2. edition.
- [Morris, 2000] Morris, W., editor (2000). *American Heritage Dictionary of the English Language*. Houghton Mifflin, 4. edition.
- [NASA, 1994] NASA (1994). Work Breakdown Structure Reference Guide. <http://www.tarrani.net/shared/WBSRefGuide3.pdf>. Online; accessed February 24th, 2006.
- [NHS - Modernisation Agency, 2002] NHS - Modernisation Agency (2002). What is Protocol-Based Care?... http://www.modern.nhs.uk/protocolbasedcare/whatis_leaflet.pdf. Online; accessed February 04th, 2006.
- [OpenClinical, 2006] OpenClinical (2006). OpenClinical.org. <http://www.openclinical.org>. Online; accessed March 15th, 2006.
- [Piccolo, 2006] Piccolo (2006). HCIL - Human Computer Interaction Lab, University of Maryland. <http://www.cs.umd.edu/hcil/piccolo>. Online; accessed April 14th, 2006.
- [Plaisant et al., 1998] Plaisant, C., Mushlin, R., Snyder, A., Li, J., Heller, D., and Shneiderman, B. (1998). LifeLines: Using Visualization to Enhance Navigation and Analysis of Patient Records. In *Proceedings of the*

- 1998 American Medical Informatic Association Annual Fall! Symposium, pages 76–80.
- [pma - Project Management Austria, 2002] pma - Project Management Austria (2002). pm baseline (English Version). http://debian.p-m-a.at/docs/pm_baseline_en.pdf. Online; accessed February 19th, 2006.
- [Project Management Institute, 2000] Project Management Institute (2000). *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*. Project Management Institute, 2000 edition.
- [Rit, 1986] Rit, J.-F. (1986). Propagating Temporal Constraints for Scheduling. In *AAAI*, pages 383–388.
- [Shahar, 2002] Shahar, Y. (2002). Automated Support to Clinical Guidelines and Care Plans: The Intention-Oriented View. <http://www.openclinical.org/docs/int/briefingpapers/shahar.pdf>. Online; accessed February 28th, 2006.
- [Sisk, 1998] Sisk, T. (1998). History of Project Management. <http://www.microsoft.com/downloads/thankyou.aspx?familyId=C1F9B881-D879-4B54-B07B-55041685F15F&displayLang=en&oRef=>. Online; accessed July 06th, 2006.
- [Smith, 1994] Smith, S. (1994). OPIS: A Methodology and Architecture for Reactive Scheduling. In Zweben, M. and Fox, M., editors, *Intelligent Scheduling*. Morgan Kaufmann.
- [SourceForge, 2006] SourceForge (2006). SourceForge GanttProject. <http://ganttproject.sourceforge.net>. Online; accessed March 15th, 2006.
- [Sun Microsystems, 2001] Sun Microsystems (2001). Programmer’s Guide to the Java 2D™ API - Enhanced Graphics and Imaging for Java. <http://java.sun.com/j2se/1.4/pdf/j2d-book.pdf>. Online; accessed April 20th, 2006.
- [@Task, 2006] @Task (2006). @Task Project Management Software. <http://www.attask.com>. Online; accessed March 15th, 2006.
- [The Standish Group, 2001] The Standish Group (2001). Extreme Chaos. http://www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf. Online; accessed February 20th, 2006.
- [Tu et al., 2002] Tu, S., Johnson, P., and Musen, M. (2002). A Typology for Modeling Processes in Clinical Guidelines and Protocols. Technical Report SMI-2002-0911, Stanford University.

- [Tu and Musen, 2001] Tu, S. and Musen, M. (2001). Modeling Data and Knowledge in the EON Guideline Architecture. <http://smi-web.stanford.edu/auslese/smi-web/reports/SMI-2001-0868.pdf>. Online; accessed March 18th, 2006.
- [Vidal, 2004] Vidal, T. (2004). The Many Ways of Facing Temporal Uncertainty in Planning and Scheduling. In *11th International Symposium on Temporal Representation and Reasoning (TIME 2004)*, pages 9–10.
- [Vidal et al., 1996] Vidal, T., Ghallab, M., and Alami, R. (1996). Incremental Mission Allocation to a Large Team of Robots. In *IEEE International Conference on Robotics and Automation (ICRA '96)*, pages 1620–1625. Aldershot.
- [Voigt, 2002] Voigt, R. (2002). An Extended Scatterplot Matrix and Case Studies in Information Visualization. Master's thesis, Hochschule Magdeburg-Stendal, Austria.
- [Wikipedia, the free encyclopedia, 2006a] Wikipedia, the free encyclopedia (2006a). Frederick Winslow Taylor. http://en.wikipedia.org/wiki/Frederick_Winslow_Taylor. Online; accessed July 06th, 2006.
- [Wikipedia, the free encyclopedia, 2006b] Wikipedia, the free encyclopedia (2006b). Namespace. <http://en.wikipedia.org/wiki/Namespac>. Online; accessed April 18th, 2006.
- [Wikipedia, the free encyclopedia, 2006c] Wikipedia, the free encyclopedia (2006c). Project Management. http://en.wikipedia.org/wiki/Project_management. Online; accessed July 06th, 2006.
- [Woolf et al., 1999] Woolf, S., Grol, R., Hutchinson, A., Eccles, M., and Grimshaw, J. (1999). Clinical Guidelines: Potential Benefits, Limitations, and Harms of Clinical Guidelines. *BMJ*, 318:527–530.
- [Wysocki et al., 2000] Wysocki, R., Beck, R., and Crane, D. (2000). *Effective Project Management*. John Wiley & Sons, Inc., 2. edition.