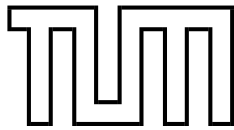
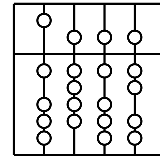


Technische Universität
München
Fakultät für Informatik



Lotus Notes/Domino
Unified Modeling Process

- Systementwicklungsprojekt -

Juni 2004

Christoph Angerer
angerer@in.tum.de · Matrikelnummer: 2292793

Aufgabensteller:
Prof. Bernd Brügge, Ph.D.

Betreuer:
Oliver Creighton

Technische Universität München · Sommersemester 2004

Special thanks to Tobias Klüpfel for all the valuable discussions

Abstract

"In too many organizations, Lotus Notes and Domino application development is out of control." [Ives1999]

Lotus Notes popularized the concept of groupware, and with it the promise of rapid application development. But, as more organizations started to embrace Notes, the demand for more sophisticated features grew and resulted in what is today a rather complex development environment.

Besides IBM, several companies offer development tools which can help reduce this complexity. However, current solutions mostly support single development tasks, such as versioning or configuration management. But the remaining lack of a clear and standardized methodology for developing Notes applications still leads to an opaque development process reserved to Notes experts.

This paper describes NoDUMP (Notes/Domino Unified Modeling Process), an extension to the UML, which enables developers to use state-of-the-art UML based software development processes and object-oriented methodologies to model Notes applications. By using UML as a basis, software developers with different areas of expertise in the field of software development can communicate and document their solutions across all phases of the development process. Additionally, the extended UML models could be used for automated code generation of Notes databases.

In conclusion, two case studies show the practical usage of NoDUMP and how it can be applied to real life projects.

Contents

1 Motivation **7**

2 Requirements Elicitation **7**

2.1 Scenario 7

2.2 Problem Statement 9

2.3 Scope of the Solution 9

2.4 Requirements 10

2.4.1 Functional Requirements 10

2.4.2 Nonfunctional Requirements 10

2.5 Use Case Model 11

2.6 NoDUMP in the Project Lifecycle 15

3 Lotus Notes Architectural Model **16**

3.1 Basic Architecture 17

3.2 Lotus Notes Design Documents 17

4 System Design **19**

4.1 Design Goals 19

4.2 Subsystem Decomposition 20

5 The NoDUMP Specification **21**

6 Case Studies **23**

6.1 Design Database 23

6.1.1 Problem Statement 23

6.1.2 Analysis 23
6.1.3 Top Level Design 25
6.1.4 Detailed Design 27
6.1.5 Implementation 28
6.2 iTrack 29
6.2.1 Problem Statement 29
6.2.2 Analysis 29
6.2.3 Top Level Design 29
6.2.4 Detailed Design 31
6.2.5 Implementation 34
6.3 Requirements Evaluation 35
6.3.1 Fulfilled Requirements 35
6.3.2 Partly Fulfilled Requirements 35
6.3.3 Unjudged Requirements 36

7 Future Work **37**

Appendices **38**

A Extending UML **38**

B List of Figures **40**

C References **41**

1 Motivation

"Lotus Domino provides a multiplatform foundation for collaboration and e-business, driving solutions from corporate messaging to Web based transactions - and everything in between." (http://www.lotus.com/domino)

Since its introduction to the market in 1989 the fundamental document-based object model of Lotus Notes/Domino has been retained mostly unchanged. While inveterate Notes developers praise this document-centric view as the perfect way for building groupware applications, software engineers who are not used to it flinch from using Notes because of the same reason. Object-oriented software engineering as a way of conquering complex and changing systems fails when it comes to Notes.

Currently, no sufficient and standardized methodology for developing Notes applications exists, even though it is often requested in Notes related newsgroups. Some developers try to use entity relationship diagrams or UML (Unified Modeling Language) to do some analysis and to define the data model for their applications. But none of these methods are capable of describing entire Notes applications.

Because of the lack of a sufficient modeling technique, the development of Notes applications is still reserved to experts. This causes a blurring of common developer roles where analysts and architects design a system while programmers and designers build the application and its user interface. During all stages of a development process, real Notes experts have to be involved.

2 Requirements Elicitation

2.1 Scenario

Actors

Actor	Description
BioNews Exchange Inc.	Company which organizes conferences, customer
SoftApps Inc.	Small company specialized on web applications
Susan	Analyst at SoftApps Inc., Notes rookie
Toby K.	Notes expert, Lotus Certified Developer

BioNews Exchange Inc. is a company which organizes conferences in the field of gene manipulated food. Because the number of participants increased steadily over the last years, BioNews Exchange Inc. wants to improve the whole participant management. Therefore SoftApps Inc. gets an order to develop a webbased software system for this task.

At a first step, Susan takes through an as-is analysis of how participants are managed within BioNews Inc. today, including registration of participants to a conference, billing of visited conferences, booking of hotel rooms, sending news to registered people and much more. Based on this information, Susan and her team write an analysis document containing the problem statement, the requirements, use cases and the application domain model.

Based on the application domain model, SoftApps Inc. decides to realize the system using Lotus Notes/Domino. The rationale behind this decision is not so much the broad functionality of Notes, which could be used, but more the structure of the application domain. It turns out, that the whole model is ruled by inheritance relationships which are more difficult to realize using a relational database management system than using Notes.

Toby, as the Notes expert of SoftApps Inc., takes the UML application domain model as the starting point to develop the system design. In a first step, he refines and annotates the model conforming to the NoDUMP UML Profile and uses the Model-View-Controller paradigm to design the user interface and functional elements. The automated generation transforms the model into a partly-functional Notes database skeleton at every time.

During the whole development process, Toby as the Notes expert and Susan as the application domain expert, who does not know Notes in detail, can talk about the solution on a strictly UML basis. The generated prototypes are used to validate the system under development referring to the Requirements Analysis Document and BioNews Exchange Inc. officials.

After the successful client acceptance test, the whole system is handed over to BioNews Exchange Inc. Together with the binaries, a complete documentation of the developed system is delivered. This documentation includes the annotated UML diagrams which describe all aspects of the Notes solution, supporting future changes and maintenance.

2.2 Problem Statement

In contrast to object-oriented software design patterns, within Notes, the data schema, the user interface, and parts of the business logic are all intended to be packed together into single objects, called “Documents”. The resultant huge complexity for stand-alone Notes applications becomes even worse when Notes is to be integrated into larger software systems. The absence of a common platform for communication between Notes experts and the other developers affects the whole development process, from requirements elicitation through implementation up to documentation.

The goal of this paper is to develop a UML Profile (an extension to the UML) which is capable of describing Notes applications and which forms the basis for the integration of Notes development with state-of-the-art UML based development processes, like for example the Rational Unified Process. This extension should also be usable and understandable by developers who do not know Lotus Notes/Domino in detail. Additionally, translation rules will be elaborated which can be used in the future to develop tools which automate the generation of Notes databases from an abstract model.

2.3 Scope of the Solution

UML defines numerous diagrams and diagram elements which can be used to model different aspects of a system, from requirements over static design up to dynamic behavior. Especially the dynamic view onto a Notes application would be important, not only to model the flow of form-based web-applications but also to model functional elements like Notes agents.

On the other hand, Notes is more than just a framework for custom applications. Replication mechanisms, load balancing and LDAP services as well as mail routing functionality and integration with other application or database servers make it important to model server setup and security issues carefully in order to keep a complex server topology up and running.

Because NoDUMP is a first step to standardize modeling and documentation for Notes, NoDUMP focuses on these static structures and leaves dynamic issues for future work.

Furthermore, NoDUMP is specialized on web usage. With NoDUMP , it is not possible to express special features of the Notes client which can not

be directly used for web applications. But most of these features could be modeled using NoDUMP by adding additional information for the developers how to handle them (e.g., “realize this ’View’ as a ’Notes Folder’ ”).

2.4 Requirements

2.4.1 Functional Requirements

MVC Paradigm: Because Notes is commonly used to realize interactive applications, the Model-View-Controller design must be supported by NoDUMP .

Fit into Common UML Processes: NoDUMP must fit into state-of-the-art engineering processes. That is: Models describing Notes specific parts can be used together with models describing other subsystems of the application. Standard UML models can be refined later to be implemented in Notes.

Server Topologies: It must be possible to model server topology issues with NoDUMP , like replication and mail routing within a Notes cluster. Additionally, database dependencies like design inheritance or inter-database relationships must be describable in order to plan deployment of new applications and to document collaboration of servers and databases.

Code Generation: A NoDUMP compliant model must be detailed enough to serve as an input for automatic generation of Notes database skeletons.

UML 2.0 Compliance: NoDUMP must be compliant to the UML 2.0 specification described in [UML2Infra]

2.4.2 Nonfunctional Requirements

Focus on UML Users: Models compliant to NoDUMP should be understandable for users who are familiar with UML and do not necessarily know Notes in detail.

Default Meanings and Values: NoDUMP compliant models must be detailed enough to generate meaningfull Notes database skeletons. At

the same time, the models should be understandable and easy to use. Therefore, default meanings and values for all stereotypes and tagged values should be provided wherever possible, in order to be able to leave out unnecessary information.

Support Notes Integration: Notes applications can be used to realize subsystems of larger software systems. Therefore, it should be possible to model interfaces between Notes and Non-Notes systems explicitly to gain a clear system boundary.

Support Design Patterns: It should be possible to develop and document Notes specific design patterns with NoDUMP .

2.5 Use Case Model

Figure 1 shows the use case model for NoDUMP when used within a surrounding software development process, like for example the Rational Unified Process (RUP) as described in [RUP].

Because NoDUMP is realized as an extension to the UML, Notes application development can be integrated into all UML based software engineering processes. But of course, some general activities during developing with NoDUMP can be discovered, independently from the surrounding process. The different degrees of Notes-related detail within single activities should be considered when choosing a process or planning a project.

2.5.0.1 Description of the Developer Roles

Requirements Engineer: Gathers and documents requirements together with the **Application Domain Expert**.

Application Domain Expert: An expert who knows the application domain in detail. Usually, the **Application Domain Expert** is on customer side.

Analyst: Analyses the requirements and develops an analysis model including the UML application domain model.

Software Architect: The **Software Architect** designs high level software architecture and has at least some experience in using UML.

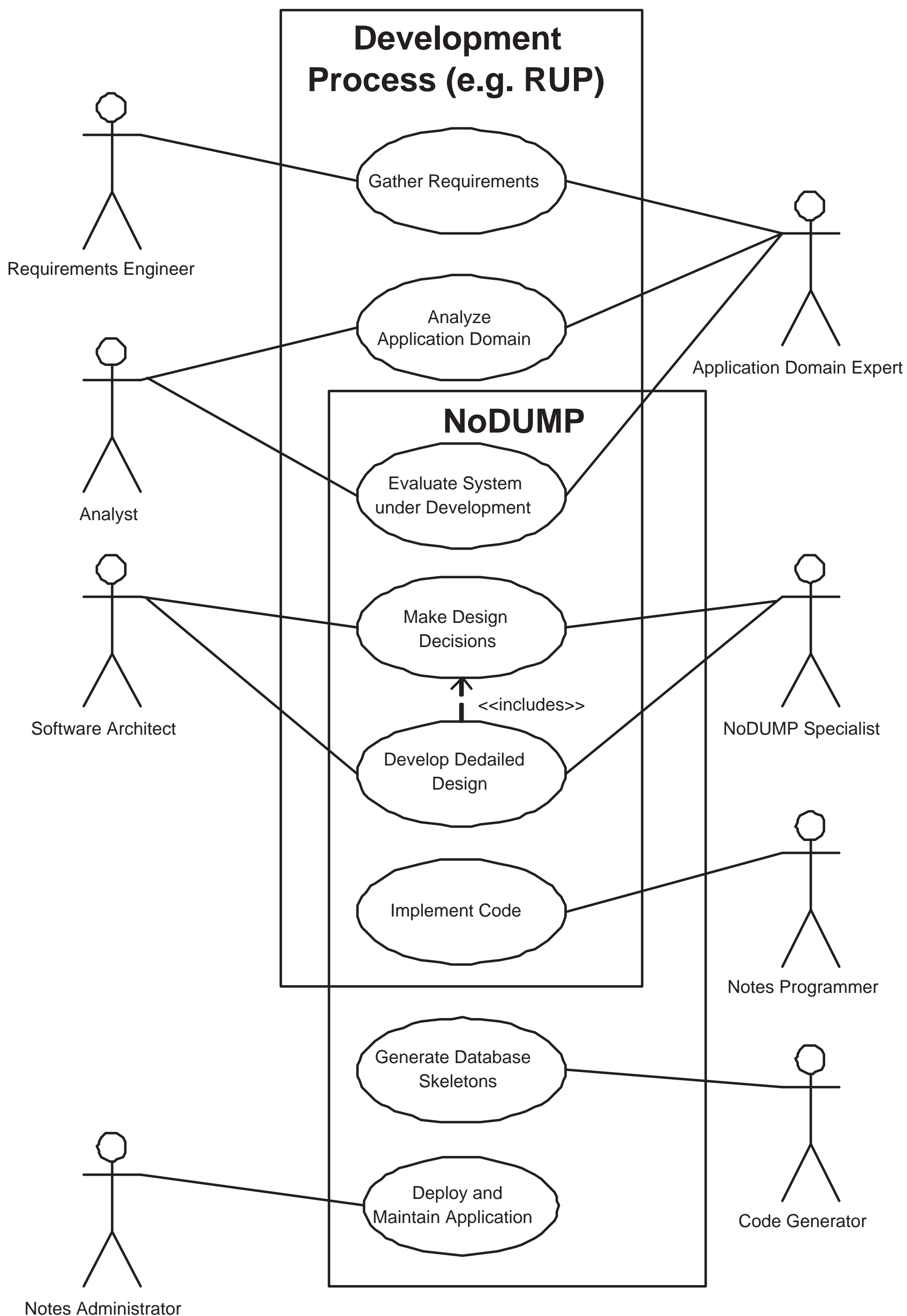


Figure 1: The Use Case Model

NoDUMP Specialist: A developer who knows how to use NoDUMP for developing Notes Applications. He does not necessarily know Notes in detail.

Notes Programmer: A developer who is skilled in programming Notes applications.

NoDUMP Code Generator A software tool similar to a compiler which takes a NoDUMP model as an input and generates Notes database skeletons.

Notes Administrator: The **Notes Administrator** administrates and maintains the Notes server and deployed applications. Usually the **Notes Administrator** is employed on the customer side.

2.5.0.2 Description of the Use Cases

Gather Requirements: The goal of requirements gathering is to express what the proposed system should do. For this task, **Requirements Engineer**s usually work together with **Application Domain Expert**s. Details on the requirement gathering phase are defined by the surrounding software development process.

Analyze Application Domain: Analysis of the application domain is the process of examining the requirements and making a conceptual model of the system to be built. For this task, **Analyst**s usually work together with **Application Domain Expert**s. Details on the analysis phase are defined by the surrounding software development process.

Evaluate System under Development: The system under development has to be evaluated by the software developers and the customer on a regular basis. The frequency of evaluations and their participants are defined by the surrounding software development process. Usually, at least **Analyst**s and **Application Domain Expert**s participate in these meetings. As an evaluation basis, working prototypes of the Notes applications, possibly generated out of the NoDUMP model, as well as the NoDUMP model itself are used.

Make Design Decisions: During developing detailed designs, design decisions have to be made. These design decisions include architectural

issues as well as definition of subsystem boundaries. **Software Architect**s will make these decisions together with **NoDUMP Specialist**s. Collections of Notes specific software patterns expressed with NoDUMP can speed up finding possible solutions for common design problems. Interfaces between Notes applications and other software systems are documented with NoDUMP .

Develop Detailed Design: The goal of this use case is to develop a detailed design model out of the analysis model to make the system realizable in software. In case of Notes applications, a NoDUMP compliant model has to be developed. This task is realized by **NoDUMP Specialist**s together with **Architect**s. The single activities and their contents are defined by the surrounding process.

Implement Code: Generated Notes database skeletons have to be completed with code which could not be generated out of the NoDUMP model. This task is performed by **Notes Programmer**s who use the Lotus Notes/Domino development tools and refer to the analysis documents as well as the NoDUMP model.

Generate Database Skeletons: The generation of Notes database skeletons from NoDUMP compliant models is done by a **Generator**. This task includes validation of the model with reference to the NoDUMP specification and transforming it into a Notes binary format or the "Lotus Domino XML" format.

Deploy and Maintain Application: After a successful client acceptance test a **Notes Administrator** is responsible for deploying and maintaining the Notes applications. The **Administrator** uses NoDUMP to document the concrete deployment and collaborations of the different databases and servers. For maintenance issues, the **Administrator** can refer to the documentation of the database which includes the detailed NoDUMP models.

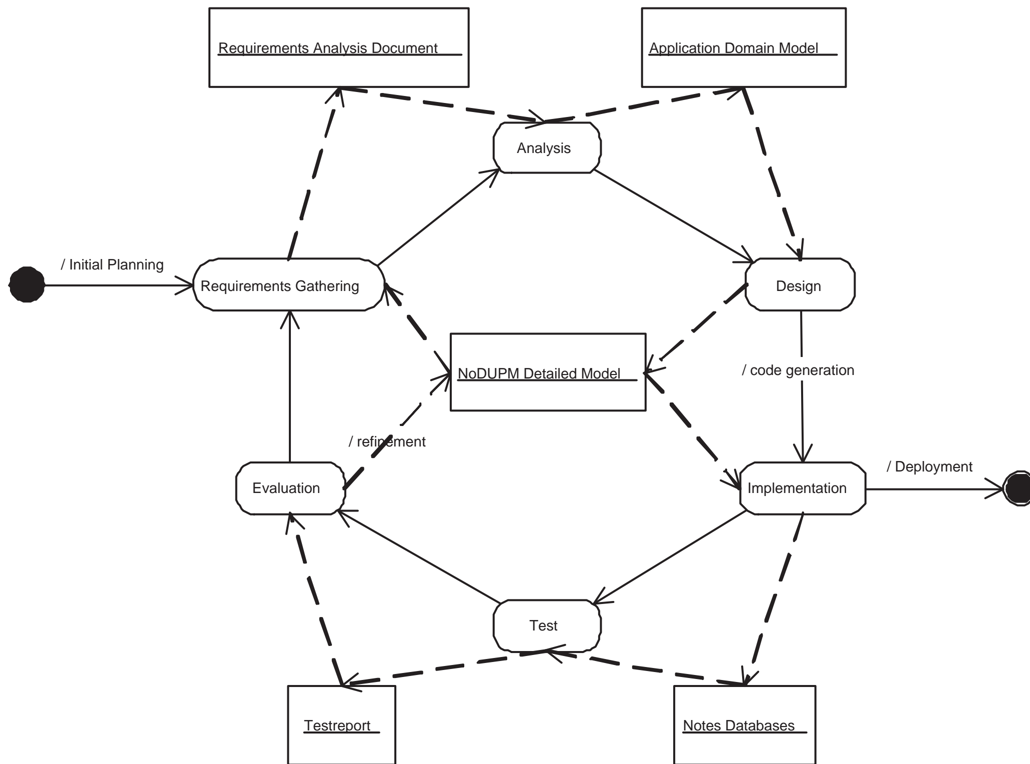


Figure 2: Integration of NoDUMP into RUP

2.6 NoDUMP in the Project Lifecycle

The integration of NoDUMP into an iterative process is shown in Figure 2. The phases correspond to the Rational Unified Process phases described in [RUP].

During the design phase, a detailed model of the application is developed. In case of Notes development, this model is compliant to NoDUMP , so a code generator can be used to generate Notes database skeletons. These skeletons will be implemented during the implementation phase.

After testing the current system under development, the test-results will be evaluated against the requirements analysis document. During the evaluation phase the NoDUMP detailed model may be refined and functions as an input for the next iteration.

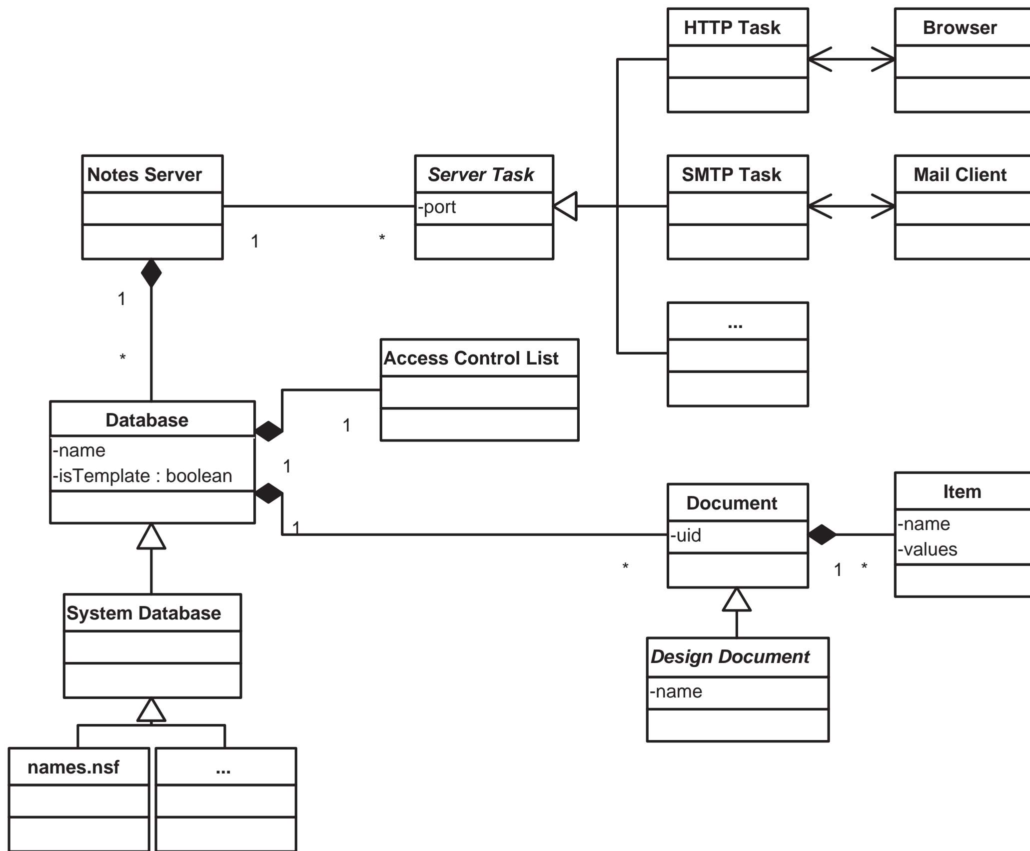


Figure 3: Lotus Notes Architectural Model

3 Lotus Notes Architectural Model

This chapter gives an overview over the most important entities within Notes. Figure 3 shows the basic architectural model of Notes and figure 4 shows a conceptual model of the design documents provided by Notes. Both figures will be explained in the following two sections.

3.1 Basic Architecture

The **Notes Server** is the central application which manages all databases and server tasks.

For communication with other software systems, several **Server Task**s can be started. Each server task realizes a certain protocol, e.g., for communication over HTTP, SMTP or LDAP. A task usually accesses the databases on the server, processes the request and generates an appropriate response for the client.

The principle building block of a Notes application is the **Database**. But a Database does not only contain data, as the name may imply, but also holds the business logic and design elements. Therefore, a Database is usually an entire application. The access rights to a Database are defined by an **Access Control List**.

The Notes server itself heavily uses databases for realizing its own management tasks. These **System Database**s are used for server configuration, user management, error logging, mailing purposes and so on.

The structure in which a Database stores its data is called **Document**. A unique id is automatically assigned to a Document on creation. In contrast to "record sets" used by relational database management systems, a **Document** does not define any schema for its data. Instead, a Document uses name-values pairs which can be inserted dynamically.

3.2 Lotus Notes Design Documents

Design Documents hold the data schemas, business logic, and user interface (the "source code" of a Notes application). As shown in figure 3, Design Documents are similar to any other Document storing application data. Therefore, all mechanisms like replication or versioning provided by the Notes Server can be used for the source code itself. This enables the developers to distribute application changes over several database instances and even Notes Servers.

Figure 4 shows the most important Design Document types provided by Notes as well as their conceptual dependencies. Overall, Notes provides 12 different types of Design Documents for different purposes.

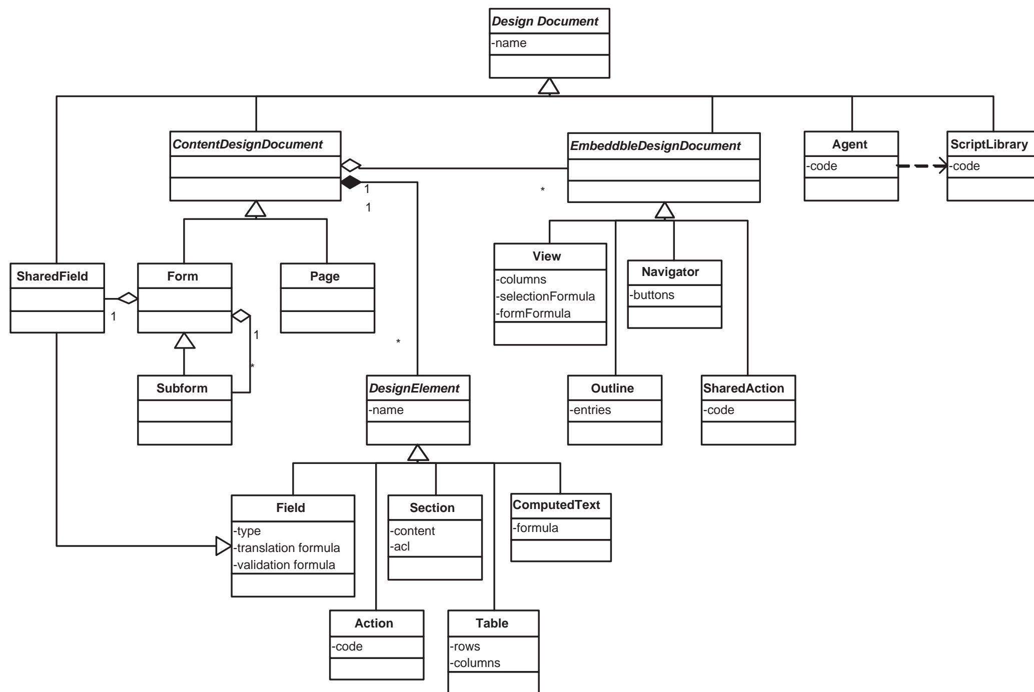


Figure 4: Lotus Notes Design Documents Model

DesignElements can be used within **ContentDesignDocument**s for layout and functional purposes, but a **DesignElement** is no **DesignDocument** by itself and can therefore not be replicated. Examples for **DesignElement**s are **Table**s for layouting or **Action**s, similar to HTML-links or buttons, for executing server side code.

EmbeddableDesignDocuments are concrete **DesignDocument**s but they can only be displayed to the user by embedding them into a **Content-DesingDocument**. **Navigator**s and **Outline**s are used to realize different types of user menus. **SharedAction**s define a piece of code which is executed when a button or link is pressed. **View**s are used to select a subset of all existing documents of a database and present them in a tabular style. Which documents should be included within a view is defined by a "selection formula", similar to an SQL statement. If a **View** should be directly displayed to a user, either a Notes default form or a customized view template form is used, so a **View** is alway embedded into a **Form** when presented.

The user interface is realized by **ContentDesignDocument**s . They can include text, pictures, **DesignElement**s as well as **EmbeddedDesignDocument**s to present content. **Page**s are used for static content, similar to

HTML pages. **Form**s create or display dynamic content stored in the **Documents** of a **Database**. **Field**s or **SharedField**s access the corresponding values of the **Document** for presentation or editing. **Field**s and values are associated by name equality. **Subform**s are very similar to **Form**s but they can only occur within a surrounding **Form**.

Most of the business logic of Notes applications is realized within **Agent**s. An **Agent** is a piece of code which can be executed on certain events. These events may be triggered by the user (e.g., by pressing a button) or by system events (e.g., on schedule). Code which is used by multiple **Agent**s or **Action**s can be separated into **ScriptLibraries**.

4 System Design

4.1 Design Goals

The NoDUMP Profile extension to the UML has been designed with the following design principles in mind:

Readability of Diagrams While it is possible to include loads of Notes-related information into the diagrams, this is not strictly necessary. Detailed information can be left out without changing the meaning of the models. Together with context-related default stereotypes the readability of the resulting diagrams can be maximized.

Components for single Tasks Each component of NoDUMP can be used to design and model different aspects of an application, like for example data model, business logic or user interface. This allows to define developer roles and responsibilities and supports a clear structure within Notes Applications.

System Boundary In order to gain a well-defined interface between Notes databases and external applications NoDUMP cannot be mixed with standard UML elements or other UML profiles within one package. In case non-stereotyped UML elements are used within a NoDUMP model, default NoDUMP stereotypes are assumed.

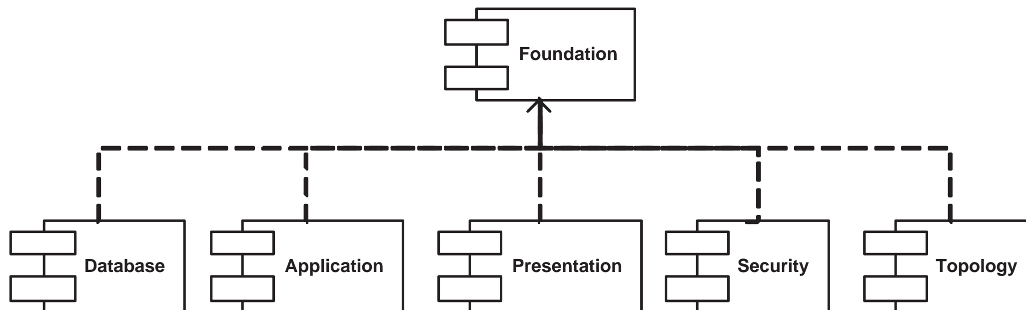


Figure 5: The components of the NoDUMP Profile

4.2 Subsystem Decomposition

The Standard elements are defined within a single package called NoDUMP-Profile. The specification of the NoDUMPProfile package defines stereotypes to model Notes related applications. Because of its size, the whole profile is divided into single components, each to model different aspects of an application.

Figure 5 shows all components of the NoDUMP Profile.

Foundation: Abstract stereotypes used by other components.

Database: Stereotypes used to model the database functionalities of Notes.

Application: Stereotypes used to model business logic realized with Notes.

Presentation: Stereotypes used to model (web) user interfaces driven by Notes.

Security: Stereotypes used to model security aspects of Notes.

Topology: Stereotypes used to model topology issues (like deployment of databases over several servers) of Notes.

5 The NoDUMP Specification

This section gives a short overview over the NoDUMP specification which has been developed during this project. The specification consists of the following main sections:

1. **Motivation:** Describes the motivation, the problem statement, and the scenario.
2. **Approach and Structure:** Gives explanatory notes on extending UML, the design goals, and the structure of the whole specification.
3. **Lotus Notes Architectural Model:** An introduction to the conceptual model of Lotus Notes.
4. **The NoDUMP Profile:** Specifies the stereotypes, tagged values, and constraints for the NoDUMP Profile. This section is divided into the following subsections:

   Profile Specification Overview: Provides UML related context information of the NoDUMP Profile and specifies the applied naming conventions.

   Subsystem Decomposition: Gives an overview over the whole profile which is logically divided into several components.

   Component Specifications: Specifies all stereotypes of each component in detail.
5. **Translation of NoDUMP Models:** Describes the most important rules for translating NoDUMP compliant models into Notes database applications. The rules are given in a graphical notation using the profile as well as instances of Notes objects introduced in section 3.
6. **NoDUMP Development Process:** The integration of NoDUMP into state-of-the-art UML-based development processes is described in this section. As a representative process the Rational Unified Process (RUP) is used.

To support the expected usage of the specification as a reference book, each subsection explaining a single component of the profile is organized similarly. Each component description consists of:

an Introduction: Each component is introduced giving a short description.

a Diagram: A UML diagram visualizes the abstract syntax of each component (i.e. the classes and their relationships) together with some of the constraints (multiplicity and types). These diagrams use stereotypes defined by [UML2Infra] for extending UML.

the Stereotype Descriptions: All stereotypes are listed in alphabetical order. A short description for each stereotype explains its semantic and additional constraints.

the Data Type Descriptions: All data types which are defined in a component are explained in detail.

some Examples: One or more graphical examples show possible usages of a component.

The specification of the NoDUMP UML Profile reuses some chapters of this paper, even though they are presented in a different order to adhere to a common specification style (similar to [CORBA UML Profile]).

6 Case Studies

To evaluate NoDUMP and to provide further examples, two Notes databases have been realized: The **Design Database** as a library that defines design and user interface elements which can be used for other applications. **iTrack** on the other hand is an application similar to a bulletin-board which uses an issue model proposed by [Dutoit 2002] and furthermore demonstrates the usage of the **Design Database**. In the following, short outlines of the development documents are given.

6.1 Design Database

6.1.1 Problem Statement

Most Notes based applications which are used within a single company define their own user interfaces. This often causes a nonuniform look and feel of the single applications over the time and aggravates maintainability. To keep a consistent corporate identity, speed up application development, and support design changes in the future a single library which encapsulates the look and feel of Notes applications should be developed.

6.1.2 Analysis

Each element of an application's user interface can be assigned to one of the following three scopes:

Corporate Identity Common elements which support navigation within the company's web page and define the basic look and feel of all applications (see figure 6).

Application Identity Elements which are direct parts of the developed application. These include elements for navigation within the application, using the application as well as defining application related design, like for example an application logo (see figure 7).

Instance Identity Elements which can be used to customize the appearance of an application when instantiated for a project (see figure 8).

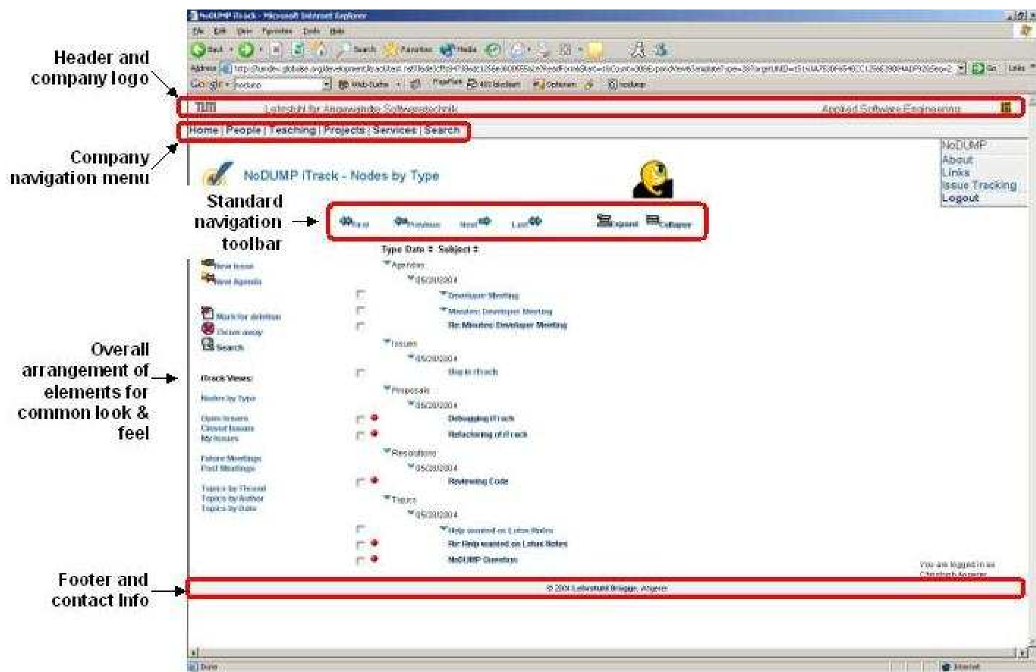


Figure 6: Corporate Design Elements

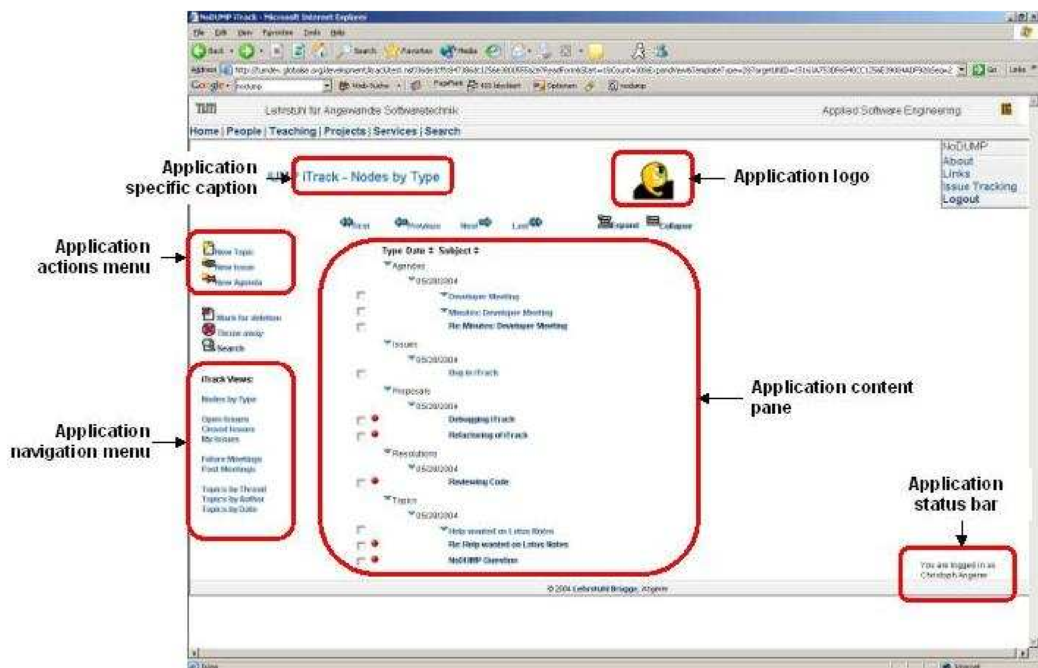


Figure 7: Application Design Elements

Figure 8: Project Instance Design Elements

Depending on the scope a design element belongs to, the frequency as well as the responsibility for creating or changing the element usually differs.

Scope	Responsibility	Frequency
Corporate Identity	Management	rarely
Application Identity	Developers	usually once for each application
Instance Identity	Administrator	usually once for each instance

6.1.3 Top Level Design

Figure 9 shows a draft of the components and their interfaces for the design database.

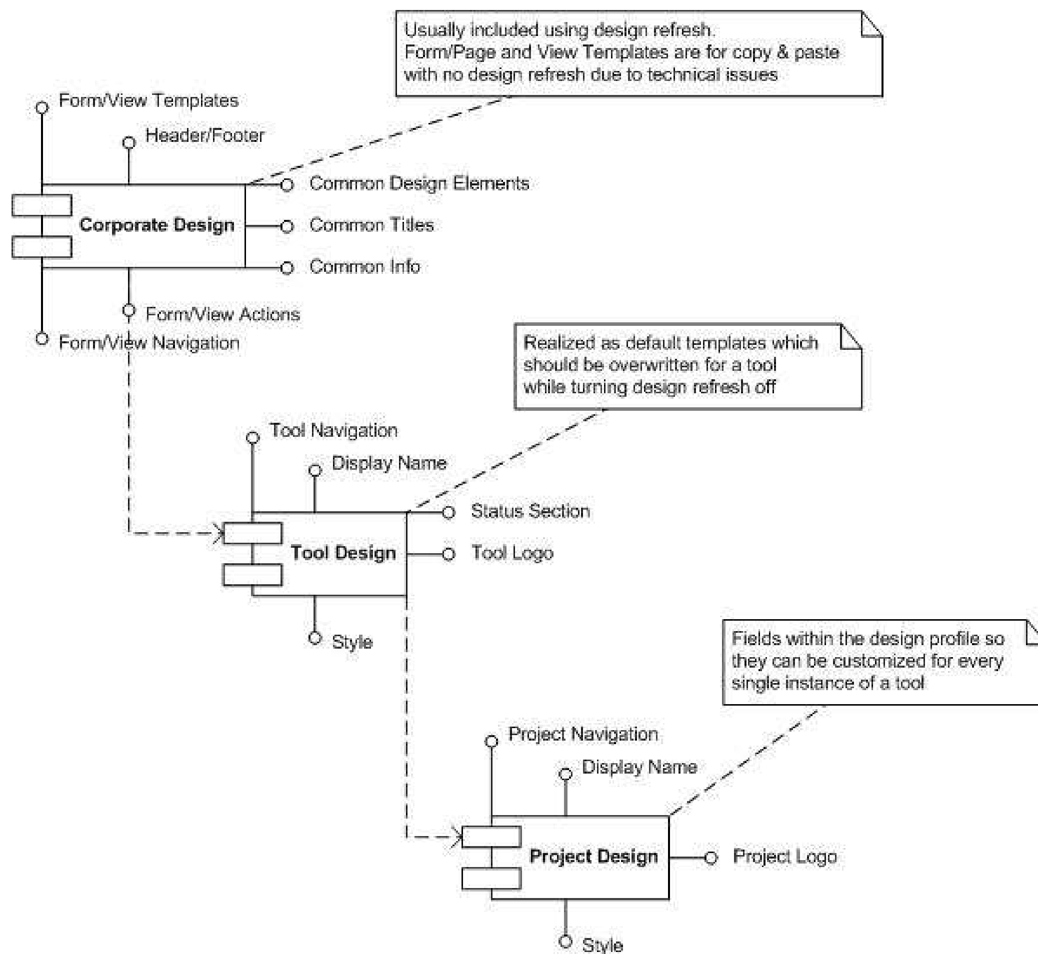


Figure 9: Draft of Design Database Components

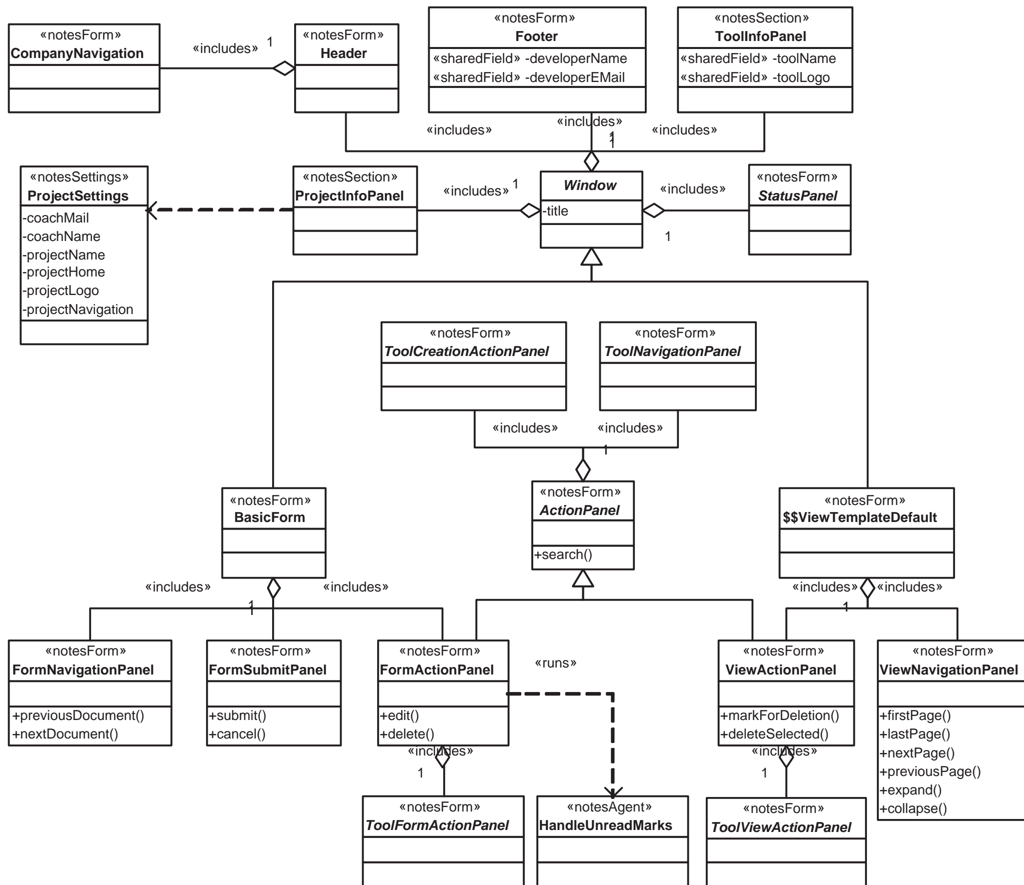


Figure 10: Detailed Design of the Design Database

6.1.4 Detailed Design

Figure 10 shows the detailed object design of the design database using the NoDUMP UML extension.

The **Window** defines the basic arrangement for all rectangular sections of the page. It includes the **Header** containing the **CompanyNavigation**, the **Footer**, a **ToolInfoPanel** for displaying the application's name and icon, an application related **StatusPanel**, and the **ProjectInfoPanel** which presents customizable information for single instances of the database.

The **BasicForm** will be used by applications as the basis for single Notes forms. It includes standard panels for common user tasks like navigating to the previous or next document, editing or deleting the document, and sub-

mitting or cancelling changes. The **FormActionPanel** can be customized by application developers to provide control elements for creating new application specific document types (**ToolCreationActionPanel**), navigate within the application (**ToolNavigationPanel**), and to provide further functionalities within a form like sending an email (**ToolFormActionPanel**).

The **$$ViewTemplateDefault** defines the default layout for presenting Notes views. Analogical to the **BasicForm**, the **$$ViewTemplateDefault** includes panels for presenting certain user menus.

6.1.5 Implementation

The detailed model of the design database has been transformed into a Notes database template by applying the rules described in **??**. Due to the lack of an automated generator the translation has been done manually.

The resulting template is located in a Notes database template called ”libs/ design.nsf”. To gain further flexibility, the abstract class ”Window” has been broken up by directly adding its functionality to the child classes ”BasicForm” and ”$$ViewTemplateDefault”. So the overall layout of Notes views and Notes forms could be made completely different in future, if needed.

An application can inherit the design of the ”libs.design” database and customize tool related elements. The customization is done by overwriting the application related Notes shared fields for applying information like ”ApplicationName” and ”ApplicationLogo” and by overwriting the application related subforms to provide functionalities like navigation within the application and status information about the application. Additionally a tool can directly use common features like support of unread marks for all types of documents without any additional work.

An example for a resulting application layout can be seen in the iTrack screenshots (figure 14).

6.2 iTrack

6.2.1 Problem Statement

Project related communication is often realized using multiple applications. For example, bulletin boards, meeting planner and bug tracker come into operation. In order to simplify finding relevant information and to gain synergy effects through combined information it would be desirable to have one application which allows to manage all of the named tasks.

6.2.2 Analysis

Common communication tasks during the life time of a project are discussion of general topics, organization of meetings, documentation of meeting minutes, announcement of news for project members and tracking bugs and other issues.

Therefore, a bulletin board like application which allows different types of postings, supports searching information, automates email notification of project members and enables users to link different posts together is assumed as an ideal type of application to solve these communication tasks.

6.2.3 Top Level Design

The issue model proposed by [Dutoit 2002] defines an abstract object model for issue based software development. An excerpt of this model is shown in figure 11.

An **Issue** is the base class for all objects which realize a certain issue type, like for example ”question”, ”problem” or other topics of interest. Every **Issue** can be supplemented by **Comment**s so users can discuss about it. Possible solutions can be added to one or more issues as **Proposal**s. Each **Proposal** is evaluated against several **Criteria** using **Assessment**s. If the developers decide for a solution, an **Issue** can be closed by associating **Resolution**s with it which refer to former proposals.

Due to its flexibility, this model is used as the basis for the iTrack application which will be realized as a Notes database application.

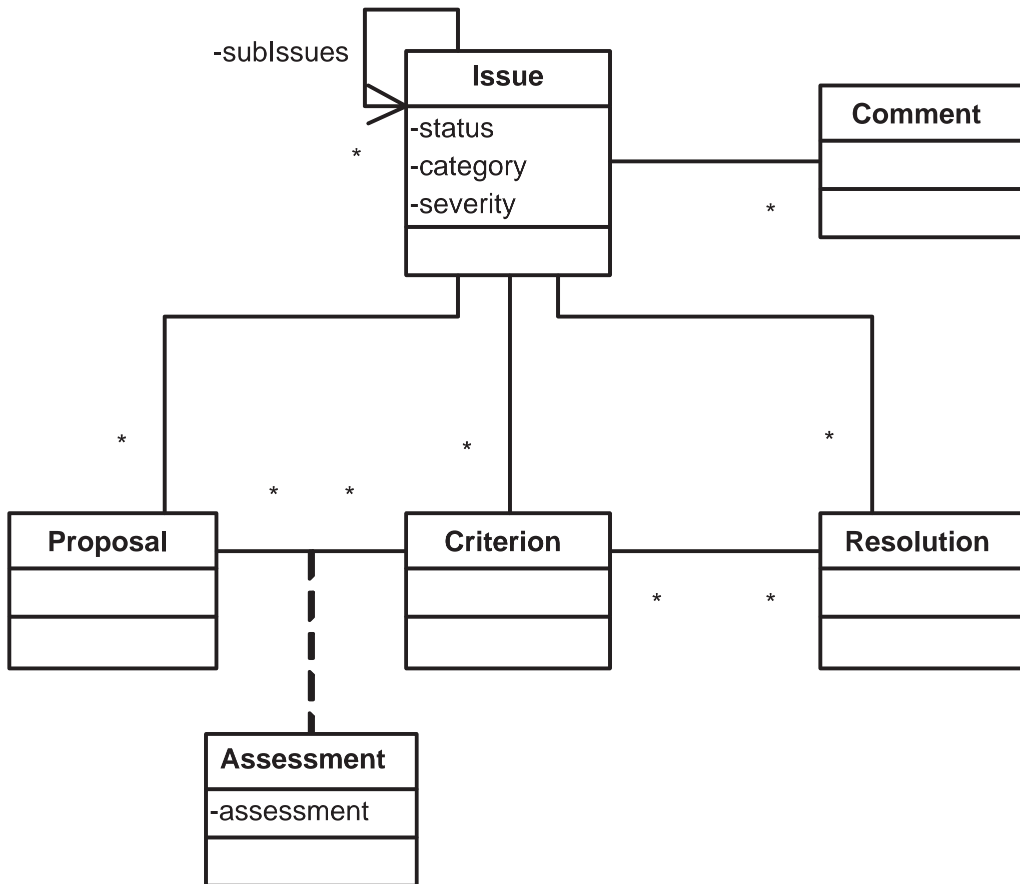


Figure 11: The Issue Model

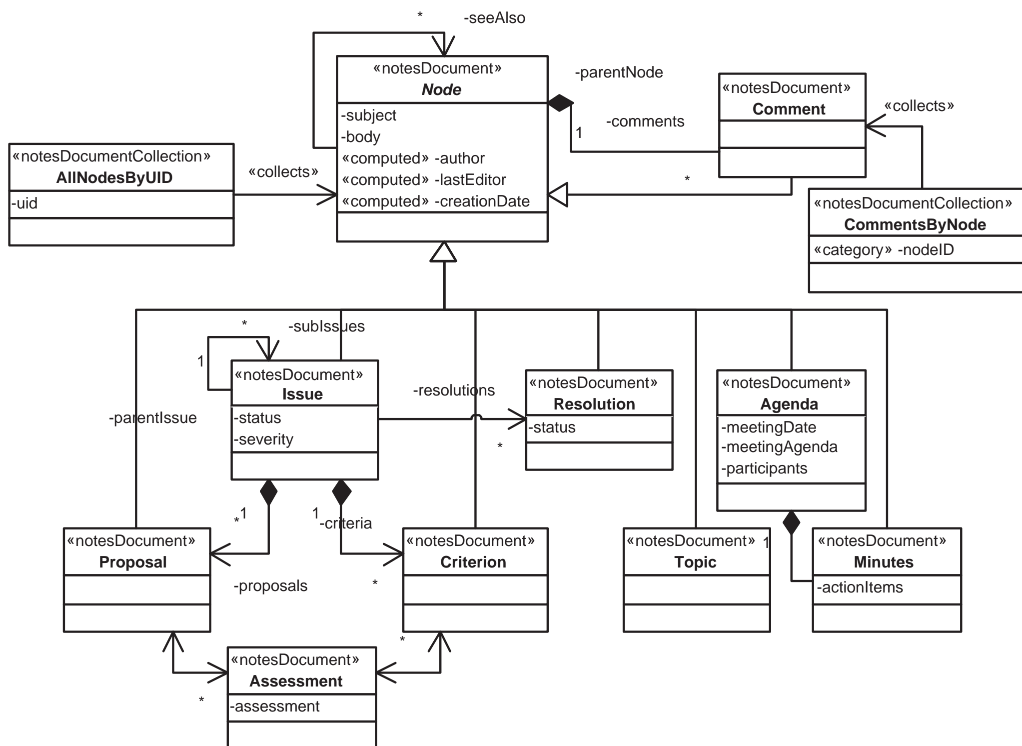


Figure 12: The **apps.itrack.model** package

6.2.4 Detailed Design

The whole iTrack application is located in the package **apps.itrack**. Within **apps.itrack**, the package **apps.itrack.model** (see figure 12) defines the model elements according to the issue model while the user interface is packed in **apps.itrack.view** (see figure 13).

The original issue model shown in figure 11 has been simplified by exchanging the many-to-many associations with one-to-many compositions (see figure 12). This step was necessary because it is not possible to create a usable HTML-user-interface for handling these complex circular relationships. Furthermore, a one-to-many composition can be directly realized using a special Notes construct called ”response documents” which simplifies implementation.

Because iTrack should support additional types of postings, like meeting agendas and meeting minutes, a super class **Node** has been added. A **Node** contains a **subject** and a **body** and automatically computes the **author** on

Figure 13: Excerpt of the **apps.itrack.view** package

creation and the **lastModifier** when a **Node** is edited. For each **Node**, **seeAlso** references to other **Node**s can be given. Additionally, **Comment**s can be attached to every **Node**. Because a **Comment** is also a **Node**, **Comment**s can be commented, too.

Figure 13 shows an excerpt of the user interface model. An abstract **NodeForm** class defines the layout for presenting attributes of a **Node**. Furthermore, every **Node** runs the **FormFeedbackController** before saving which validates the data and gives detailed feedback to the user when data is missing or invalid. **NodeForm** extends **BasicForm** from the **libs.design** template database described in section 6.1. For each concrete class of the **apps.itrack.model** package at least one **NotesForm** stereotyped class defines the individual layout. Figure 13 shows the **IssueForm** used for creating and presenting **apps.itrack.model.Issue** objects.

In a similar way, several **NotesView**s are modeled. These views present existing documents to the user in different ways like showing all unresolved issues or showing issues categorized by subject or author.

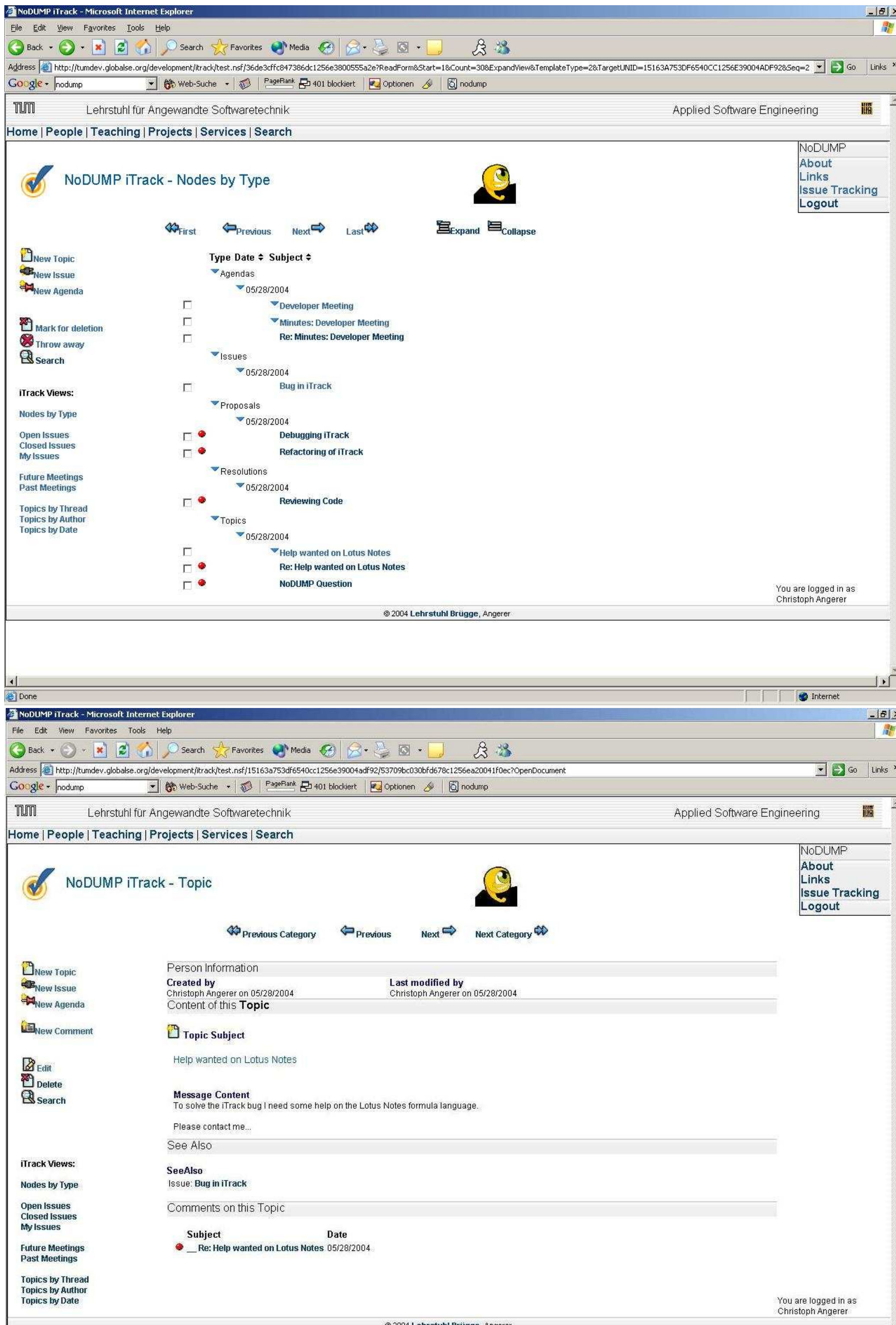


Figure 14: Screenshots of the iTrack application

6.2.5 Implementation

The detailed model of the iTrack application has been transformed into a Notes database template by applying the rules described in **??**. Due to the lack of an automated generator the translation has been done manually.

Both packages are implemented in corresponding Notes template databases called **apps/itrack/model.nsf** and **apps/itrack/view.nsf**. A third Notes template database **apps/itrack/main.nsf** inherits all design documents of the other two databases and is the template which can be used to instantiate iTrack for single projects.

apps/itrack/model.nsf holds all Notes forms and Notes subforms which define the data object model of iTrack. Within all forms each field is amended by a standard ”Input Validation” formula checking for type correctness and multiplicity of values.

apps/itrack/view.nsf realizes the graphical user interface for iTrack. Object references of the model are usually realized by ”Comboboxes” or ”Listboxes” which allow the user to select one or more objects of a certain type to be included within the reference. Forms which can be used to create new objects include a ”computed when composed” field called ”form” with the fully qualified object name of the corresponding model object as a constant value.

All GUI elements inherit their design from the **libs.design** database where all application related subforms and shared fields were overwritten as described in section 6.1.

The reusable **FormFeedbackConrtoller** controller calls the Notes ”computeWithForm” operation which evaluates the document according to the ”Input Validation” formulas of the model. All errors while validating the fields are stored within a hidden field ”exceptions”. If this field is not empty after validation the controller prohibits saving the document and displays it again using the former presentation form. Additionally, error messages are provided next to the invalid fields to enable the user to correct wrong or missing data.

Some screenshots of iTrack are shown in figure 14

6.3 Requirements Evaluation

With regard to the requirements (see 2.4) this section describes the lessons learned during the work on the case studies.

6.3.1 Fulfilled Requirements

MVC Paradigm: The MVC paradigm had a strong influence on the development of NoDUMP . Due to the design of the **Database**, **Application**, and **Presentation** components, each part can be modeled separately, even by different developer teams. By using reusable controller agents which automate validation of user inputs and give helpful user feedback (e.g., the ”FormFeedbackController” used in iTrack), the quality of the user interface can be increased without any effort.

Fit into Common UML Processes: The issue model used for the iTrack case study was treated as the application domain model which resulted from an analysis phase within a common software development process. This standard UML model was refined constantly using NoDUMP while Notes database skeletons were generated (manually) from the different states of the model. At all time, the NoDUMP model functioned as a basis for discussions and reviews with other developers. NoDUMP fitted seamlessly into this RUP-like process.

6.3.2 Partly Fulfilled Requirements

Default Meanings and Values: Default stereotypes and, even more important, default values for tagged values allow to leave out a lot of information and therefore help cleaning up the diagrams. But complex diagrams still become difficult to survey. Specifying alternate graphical notations for stereotypes could help improving readability.

UML 2.0 Compliance: NoDUMP is compliant to UML 2.0 as far as the author can tell. But for a complete specification, additional work on the single stereotypes has to be done. Well -formedness rules as well as detailed semantics have to be specified more carefully, possibly using OCL (Object Constraints Language).

Focus on UML Users: It showed, that users who are used to reading UML diagrams have no problem with understanding NoDUMP models. But

if diagrams become more complex the users need at least a basic understanding of how Notes works, especially when a lot of ≪notes-DocumentCollection≫stereotypes are used. The latter problem could be solved by modeling **NotesDocumentCollections** as operations of **NotesDocuments**, similar to the ”Home” interfaces used by Enterprise Java Beans.

6.3.3 Unjudged Requirements

Server Topologies: No server topology was modeled during the case studies.

Code Generation: For the case studies, the code was generated manually using the translation rules described in **??**. Basically it should be possible to develop a generation tool, but additional work on the translation rules has to be done before.

Support Notes Integration: During the case studies no external systems were used.

Support Design Patterns: During the work on the case studies, certain constructs appeared several times. These constructs are good candidates for becoming patterns in the future. But because no special work for finding patterns was done, this requirement could not be judged.

Additionally, some Notes related issues showed up which affect the usage of NoDUMP . On the one hand, the length of Notes DesignDocument names can only be up to 32 (64 in Notes views) characters and varies depending on the type of DesignDocument. Developers should pay attention to this restriction when a lot of hierarchical packages are used. On the other hand, the Notes formulas ”@GetDocField” or ”@DBLookup” which are used for following relationships between Notes documents can not be used within Notes view columns. This makes it hard to implement a scenario like this: ”A view presents all cars (a NotesDocument) and additionally displays the name of the owner for each car (another associated NotesDocument)”. This issue could be solved by caching the name of the owner in the ”Car” document, but then the value may not be up-to-date. Currently, there is no real solution for this problem.

7 Future Work

- Add a “behavior component” for modeling behavioral issues. Examples could be to model states of documents as state machines or to model user interactions across several forms as activity diagrams
- Complete and refine translation rules described in **??**.
- Develop a NoDUMP model validation tool which takes UML diagrams in XMI file format (XML Metadata Interchange) and validates it against the NoDUMP well-formedness rules.
- Develop a code generator which takes NoDUMP compliant UML diagrams in XMI file format and transforms it into Lotus Domino XML format according to **??**.
- Develop a refactoring tool which takes Lotus Domino XML files and transforms them into NoDUMP compliant UML models.
- Integrate NoDUMP into state-of-the-art development platforms, like for example JBuilder or Eclipse.

Appendices

A Extending UML

“Currently, there is no normative definition of a UML profile. However, the Business Object Initiative RFPs elucidated the following working definition of a UML profile.

A UML profile is a specification that does one or more of the following:

- Identifies a subset of the UML metamodel (which may be the entire UML metamodel).
- Specifies well-formedness rules beyond those specified by the identified subset of the UML metamodel. Well-formedness rule is a term used in the normative UML metamodel specification (ad/99-06-08) to describe a set of constraints written in natural language or UMLs Object Constraint Language (OCL) that contributes to the definition of a metamodel element.
- Specifies standard elements beyond those specified by the identified subset of the UML metamodel. Standard element is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value, or constraint.
- Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.
- Specifies common model elements; that is, instances of UML constructs expressed in terms of the profile.”

(from [CORBA UML Profile])

UML provides a “lightweight” extension mechanism which is supported by some UML tools. UML can be extended through:

Stereotypes The stereotype concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs. With a "stereotype" an existing UML model element can be subclassed.

Tagged values Tagged values are values which can be added to UML diagrams for modeling detailed information. Tagged values can be bound to a stereotype or be general. A tagged value is written as ≪tagname≫ .

Constraints Constraints can be added to stereotype definitions. Constraints define the context in which the "new" (stereotyped) UML Model element can be used. For example, a constraint can tell that a stereotype ≪uses≫which extends the UML element "Association" must always have a ≪person≫(which extends "Class") on its association start. While a standard association can be added between any classes, this is not true for an ≪uses≫-association, because only a ≪person≫can use something (in this example!). Constraints may be specified in OCL (object constraints language) or in natural language.

The definition of a stereotype uses the same notation as a class but it is stereotyped ≪stereotype≫. The first letter of an applied stereotype should not be capitalized. For example: a stereotype "Database" is notated as ≪database≫.

B

List of Figures

1 The Use Case Model 12
2 Integration of NoDUMP into RUP 15
3 Lotus Notes Architectural Model 16
4 Lotus Notes Design Documents Model 18
5 The components of the NoDUMP Profile 20
6 Corporate Design Elements 24
7 Application Design Elements 24
8 Project Instance Design Elements 25
9 Draft of Design Database Components 26
10 Detailed Design of the Design Database 27
11 The Issue Model 30
12 The **apps.itrack.model** package 31
13 Excerpt of the **apps.itrack.view** package 32
14 Screenshots of the iTrack application 33

C

References

[RUP] *The Rational Unified Process: An Introduction*, Addison Wesley Longman, 1999

[OOSE] Bernd Brügge, Allen H. Dutoit, *Object-Oriented Software Engineering*, Prentice Hall, New Jersey, USA, 2003.

[NotesUnleashed] Steve Kern, Deborah Lynd, *Lotus Notes and Domino 5 Development Unleashed*, SAMS, 1999.

[NotesBestPractice] *Lotus Notes & Domino - Best Practices Guide*, Lotus Development Corp., http://www.lotus.com/bpg (as Notes Helpfile)

[Ives1999] Teamstudio DesignSystem, *Software Engineering with Lotus Notes and Domino*, Ives Development Inc., White Paper, Edition 2, 1999, http://www.teamstudio.com

[MOF2003] Object Management Group, *Meta Object Facility Specification*, http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf

[UML1.0] Object Management Group, *OMG Unified Modeling Language Specification*, http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf

[UML2Infra] Object Management Group, *UML 2.0 Infrastructure Specification*, http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf

[UML2Super] Object Management Group, *UML 2.0 Superstructure Specification*, http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf

[Lotus Notes Webportal] IBM, Lotus, *Lotus Notes Webportal and Foras*, http://www.notes.net

[Dutoit 2002] S.K. Chang, Allen H. Dutoit, Barbara Paech, *Handbook of Software Engineering and Knowledge Engineering* Chapter "Rationale Management In Software Engineering", World Scientific

[CORBA UML Profile] Object Management Group, *UML Profile for CORBA Specification (based on UML 1.0)*, http://www.omg.org/docs/formal/02-04-01.pdf

[TestingProfile] Object Management Group, *UML Profile for Testing Frameworks (based on UML 2.0)*, http://www.omg.org/docs/ptc/03-08-03.pdf