High-Level Design and Analysis Of Web Applications

Ziv Yosef Shapira

Kgs. Lyngby, 2004

IMM-THESIS-2004-42

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Preface

This report constitutes my Master of Science Thesis, written during the period from January 29th to June 30th, 2004. The thesis was written at the Informatics and Mathematical Modelling department (IMM) at the Technical University of Denmark (DTU).

My supervisor has been Associate Professor Michael Reichhardt Hansen from the Safe and Secure IT-Systems group. I would like to thank Prof. Hansen for his great help, patience and assistance in writing this thesis. His insights and comments have been extremely helpful in the course of the project.

A special thanks to Jóan P. Petersen and Mikael O. Jensen for their friendly support during my entire study period.

Finally, I want to thank my wife, Betza, for her love and support throughout this busy period. To my parents, Ilana and Yoram Shapira - your help and guidance are an ongoing inspiration.

Kgs. Lyngby, June 30th, 2004

Ziv Yosef Shapira

Abstract

Design of the Human Computer Interaction (HCI), i.e. the screen structure, and Interaction Patterns, i.e. the architecture of navigation between the screens, has developed in the transition to Web applications architecture. The influence of the new architecture on the design is influenced by the technologies, such as Web browsers and communication protocols, such as HTTP. However, the methods in which this design is created or illustrated have remained basic, such as textual description or illustrative. There is also a lack of ability to relate the proposed design and the original system requirements. Furthermore we found it difficult to investigate properties of the design automatically because the description is often not formalized. Types of such properties would be: navigational, functional and architectural.

In this thesis we investigate a method both for generating effective designs of Web application navigational schemes and for describing the design. We relate between system requirements and Web Interaction Patterns (a.k.a. Navigational Design Patterns). We also introduce a software application which allows pattern designer to define patterns or combine basic patterns into more complex ones. A Web-application designer can use the tool to describe the design using patterns and investigate the design properties automatically.

A case study is introduced to demonstrate both the theoretical and the practical parts of this thesis. The current design of the system is compared to a proposed design using the application, developed during this thesis.

Keywords

Web applications, Navigational patterns, HCI, Software Design

Table of Contents

Preface....................3
Abstract....................4
Keywords....................5
Chapter 1 – Introduction....................9
Objectives.................... 11
Background.................... 12
CampusNet Example.................... 15
Contribution.................... 19
Thesis Organization.................... 20
Chapter 2 - Setting the Scene....................23
Position within the Software Design Process.................... 24
Functional Requirements.................... 25
Elements of Navigational Design.................... 27
Expressing the Design.................... 28
Reusability and Maintenance.................... 31
Diagnosis.................... 32
Chapter 3 - Navigational Design Patterns....................35
Pattern Types.................... 36
Elements of Navigational Design Patterns.................... 38
Basic Navigational Design Patterns.................... 46
Combining Navigational Design Patterns.................... 52
Summary and Discussion.................... 55
Chapter 4 - Navigational Patterns Definition Language....................57
Motivation.................... 57
Concepts.................... 59
Functional Programming - SML.................... 60
Functional Requirements.................... 62
Pattern Types.................... 64
Elements of Navigational Design Patterns in NPDL.................... 66
Defining Basic Patterns.................... 70
Defining Complex Patterns using Combination.................... 71
Instantiation of Patterns.................... 72
Diagnosis of Designs.................... 74
Summary and Discussion.................... 76
Chapter 5 - CampusNet Use Case....................79
Use Case Analysis.................... 79
Processes in the System.................... 79
Design Problems Identified.................... 80
Functional Requirements in CampusNet.................... 82

Navigational Design Patterns in CampusNet 83
Other Modules 97
Authorizations 100
Summary and Discussion 100
Chapter 6 - Visual Tool for Designers 103
Main Functionalities 103
Discussion 108
Chapter 7 – Conclusion 111
Contributions 112
Discussion 114
Conclusions 116
Further Work 117
Bibliography 119
Appendix A - Code Samples 123
Screen Components Code 123
CampusNet Design in NPDL 125
CampusNet Design Diagnosis Report 129

Chapter 1 – Introduction

It is difficult to conceive a quality software product without a well-defined and tested design behind it. Ideally, any design is based on best-practice solutions, compact and tested (as a whole and in parts) after being produced, so that it provides a solid and error-free basis for the implementation and maintenance phases. In recent years, more and more applications are implemented using Web technologies. Soon after establishing the design guidelines, it has been realized that best-practice solutions must be defined abstractly so that they can be applied to new designs which require such a solution. The first to define design patterns as means of abstraction was Christopher Alexander [AIS1]. Although patterns have started as ways of solving problems on the business logic layer of the application design (e.g. OOAD), the need for patterns on the navigational level (user interface) has soon been identified.

Web technologies have produced navigational design problems for traditional Client/Server software engineers. For example, due to the statelessness of the hypertext protocol HTTP, it is not trivial to maintain information about (or even identify) the user at each request. Another characteristic property is the user interface, which is a standard Web Browser, as opposed to a tailored user interface in client/server architecture. The user interface design has to match existing browsers capabilities rather than adapting the client to the Web application architecture. Another example would be the ability to potentially access any screen in the Web application using an HTTP request rather than following the navigation route dictated by the application's user interface. This possibility does not exist in applications based on client/server architecture and requires attention in the navigational design phase of Web

applications (for example, due to security concerns).

Although Web architecture differs from client/server architecture, it is still important to define navigational design principles. These principles match the requirements of the new web technologies on the one hand, but also retain classic software engineering concepts, since the Web application is still a software product. However for Web applications the patterns are still not fully adapted to some of the unique features of Web technologies [RSL3]. Software engineers are in the process of accumulating the set of problems that are encountered during Web-applications navigational design [RSL1]. Many of these problems are common and are encountered often, though the context may differ. This is the incentive for producing navigational **design patterns**, similar to classical software design patterns, which have existed for about a decade [GHJV].

Currently, design of the navigation in WA is based on either elaborate textual description or on graphical representation of the design [MHG1 and RSL3]. Both of these methods are problematic for several reasons. First, they are not standard which means that both implementers and other designer may have difficulty understanding the intentions of the original designer or misinterpret the intentions. Also, the ability to diagnose design properties, such as navigation schemes, authorization breach and behavior in exceptions (e.g. Wrong login or illegal input), in terms of navigation is greatly impaired in large designs due to this complex means of describing them. Another problem is the difficulty to identify which (if any) parts of the design can be reused once it needs to be extended and describe how to do this, due to the complexity of the description.

Generally, the navigational designer requires a standard and efficient (compact) way of describing the design so it is robust, extensible and reusable. It also needs to be fully comprehensible to colleagues, developers and testers.

Objectives

This thesis aims to define and demonstrate the use of a standard and compact language for defining and extending navigational design patterns. Furthermore the language is used for describing the designs of WA and is a basis for algorithms that perform properties analysis, such as can each screen be reached or which screen is most navigated through.

The project aims to combine the powerful capabilities of functional programming and the conciseness and reusability of abstract navigational design patterns to create robust Web-Application navigational designs. It is our thesis that this combination is standard, compact and provides a basis for a plethora of algorithms for diagnosis of the resulting designs.

The work carried out consists of the following tasks:

- Definition of the design patterns definition language, to allow definition of basic patterns and combinations of basic patterns to produce complex ones
- Definition of the ability to use library-defined patterns within the language to create designs of navigation within a web applications
- Create a client-server tool to demonstrate the intention
- Create an SML library of patterns and a design using the defined language
- Demonstrate a set of algorithms that diagnose the design

- A case study based on CampusNet, the DTU portal

Background

Navigational design patterns are high-level architectural abstractions that support some function in a context to answer some motivation (or problem) and provides a best-practice solution for building navigation structures that conform to this abstractions. Patterns mainly address problems that occur very often by designers. Use of patterns enhances reusability and basic parts of the design and makes it easy to switch between different implementation of the same pattern, if required [NNK1]. In other words, the idea behind the patterns is that there is no reason to "re-invent the wheel", i.e. a solution scheme (or pattern) probably exists for common problems, since the problem has been faced consistently [BCM1]. Therefore, the SW engineer needs only to adapt a well-chosen pattern to the context of the specific application's design, rather than trying to come up with a well-known and tested solution to this problem, or in a worse case, with a solution that is known to be bad or partial. For example, the "shopping basket" navigational pattern deals with a case, where users accumulate numerous objects of the same type (e.g. e-documents or books) before performing a single operation on the selected set. Such operations may be printing or purchasing. This pattern is sometimes referred to as a "Collector" [GC1]. Some other patterns are listed below:

- Sieve: how to sort users through one or more layers of choices. Used for direction through a section of the application.
- User Role: how to classify users by groups, based on behaviors, roles or permissions.
- Session: how to structure collaboration
- Virtual Product: how to display a product in a web-based catalog

As a Web-Application (WA) software engineer, ones task would be to translate a part of the user requirements into an application, based on Web technologies. The first step would be to identify the problems that the application needs to solve, based on the requirements. The next step would be to search, locate, adapt and use existing **navigational design patterns** that match these problems [GSV1]. Thus, the complete Web Information System's (WIS) navigational design shall consist of a set of navigational patterns that must address all the requirements. There are also connections between the patterns that match the various **navigational paths**; the users can follow, when using the WIS.

Another concept that needs to be added to the navigational Design pattern language is the concept of **exceptions**. As in any design, the designers' responsibility is to convey to the implementers what should happen upon a possible failure of some action performed by the user within the context of the pattern. For example, in the context of a shopping basket pattern, the operation that has been performed on the items may fail. Consequently, are there implications on what the user will experience? If so, there are implications on the design. Pattern Languages must be able to express these exceptions within the pattern description.

Another element in the overall design is the **authorization groups**. Each element, screen and path is associated with one or more authorization groups. Each user belonging to one of those groups can use the element (e.g. Button or drop-down menu), view the screen and/or navigate through the associated path.

Once basic patterns have been defined there is a need to create more complex patterns as a combination of basic patterns. This

construction method ensures the robustness and standards of the basic patterns are retained and inherited by the more complex pattern. An example would be a searchable catalog which is a combination of a search navigational pattern and a basic catalog navigational pattern. Later we discuss how this is achieved in the purposed language. Besides defining complex patterns which are also robust, their definition using basic patterns means that they inherit the current and future properties of those basic patterns automatically without the need to redefine them or update them (respectively).

A crucial part of good navigational design, apart from using patterns based on best practice solutions, is the ability to analyze the design as a whole and identify general problems as well as match it to the original requirements. The ability to do this efficiently can only be achieved if the design can be described in a standard way and can be interpreted and analyzed mechanically. The proposed language provides this capability and analysis of properties such as reachability, non reachability and navigational paths can be easily checked using predefined algorithms, which are based on the language. For example, in a complex design the designer may check that all screens that were authorized to staff members are actually reachable via the proposed design or alternatively identify screens that are on "cross roads" between several paths and should therefore be efficient. The power of standardizing the design lies in the fact that the algorithms for analysis can be written independently from the pattern and WA designers. This is because the way in which the design will be described is already known.

As a whole this set of capabilities are required by designers and researchers within the field but have not been introduced as a

whole in any one comprehensive solution, which achieves all these properties.

CampusNet Example

The use case I have chosen to follow throughout the research and development of the thesis has been CampusNet, the DTU university portal. The application itself is divided into modules, which include: login, course registration module, course participants, personal calendar and course calendar. The application is used by teachers, students and administration staff for related information retrieval and activities management.

We provide a very partial list of possible functional requirements to this application:

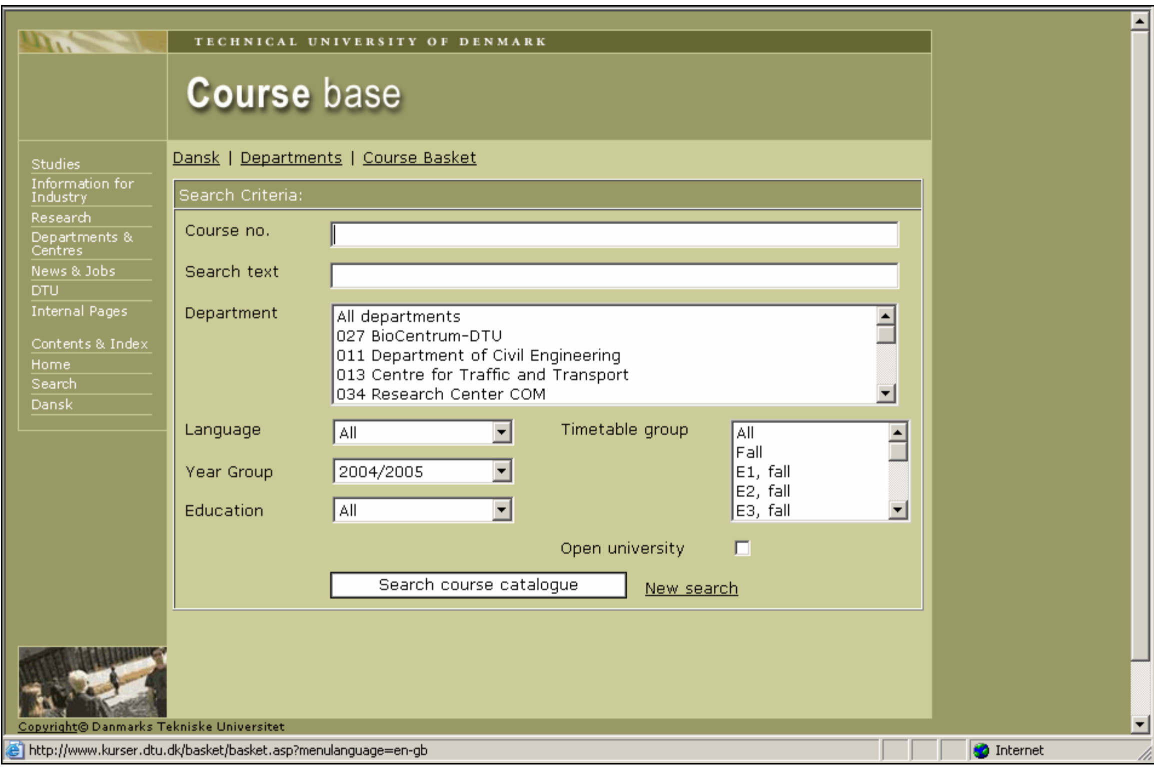
1. Enable all users to choose between Danish and English interface at any time and at any screen. The change should be in the current screen the user is viewing and any consequent screen.
2. Enables students to select courses by search criteria and register at once to all selected courses or print relevant information about them.

We use the system regularly and have identified several navigational problems in the existing design that might not match such requirements as we have presented. The problems we wish to mention are:

1. CampusNet is a bilingual application supporting both Danish and English user interfaces. The user selects the preferred language prior to logging into the system. After logging in the user could not change the language. This is a usability problem

that interferes with the user's interaction with CampusNet.

2. CampusNet enables students to register to selected courses. The courses are located in the course catalog, which is a Web application, but completely separated from CampusNet. Therefore the course selection into a shopping basket is not visible within the system. The registration process is a module in CampusNet but the selected courses need to be re-selected and also the registration is done individually to each course and cannot be done at once to a group of courses.



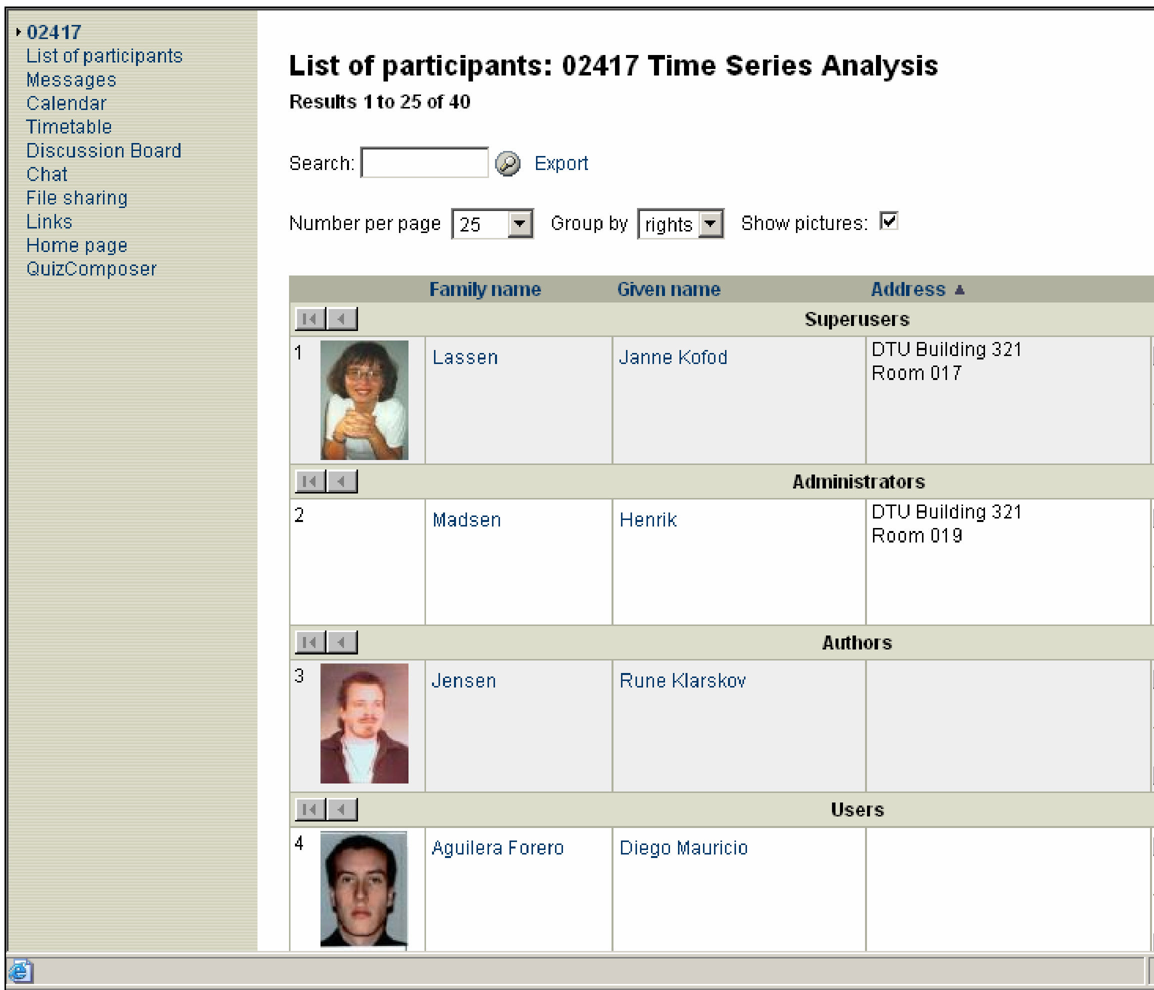
Course Catalog (external) and Course Registration module in CampusNet

Our aim is to identify design patterns and test a design using them against the actual web application. The patterns, that provide best practice solution, may have been used or we thought should have been used. Some of these patterns are:

1. The **Basket** pattern for the registration module
2. The **Calendar** pattern for “my Calendar” and “Course Calendar” module
3. The **Virtual Product** pattern for the course participant screen
4. The **Catalog with Search** for the Course Participants module

We want to show how to use these patterns in the design, but also how to create the patterns that are needed. On example would be the **Catalog with Search** pattern that uses the combination hierarchal operator ***h*** on the basic patterns: **Catalog** and **Search**.

The resulting complex pattern is then used for the design of the Course Participants module.



Course Participants module in CampusNet with the Search function

We furthermore wish to make an analysis of the design to identify whether the problems in the original application could have been identified using the proposed method. This analysis is designated for generic analysis algorithms which are not specific to this problem, but to an abstraction of problems. For example, "Where can functions be activated from?" or "Which pattern is used for a specific module?" are questions that can defined as NPDL

diagnostics and applied to any NPDL design. If the CampusNet design were to be made in NPDL, it could be found that the screen that contains the 'language selection' function is only accessible before login. This means the function cannot be used after login and does not meet a requirement to enable language change at any time. In practice this was not discovered at the design and partially fixed after implementation of the application.

We use this example to show that the goals stem from actual problems and to demonstrate the applicability of the solution described in the thesis on a real-life application.

Contribution

The goal of this thesis is to define and implement a standard way for defining and presenting navigational design patterns for Web applications and show how the patterns can be used by designer of specific applications. The work is mainly based on research of the properties that patterns need to have once defined and the way that patterns need to be extended and used in designs. These issues have been discussed in articles, such as [WV1] and [GC1].

The starting point has been research on which parts of user requirements are used by the designer of the user-interface and navigation. Once the set of requirements have been identified, a research of methods for standard designs of navigation was carried out. This is where Navigational Design Patterns have been identified and also the problems that need to be addressed in order to reach the goal of the thesis. A lack of both standardization and compactness of patterns definitions and navigational design descriptions are addressed in the thesis.

Furthermore, patterns themselves need to provide a reusable and extensible framework when defined using this language. We provide a contribution to the way in which patterns are both combined in order to create more complex patterns (e.g. Catalog with search) and reused in designs within different modules (e.g. Student calendar and course calendar). We furthermore allow additional properties to be defined for each pattern, such as authorization and exceptions, meaning what happens on the navigational level when an undesired event happens. An example of authorization would be that only users in staff can access the course management module and an example of an exception would be that a wrong password was entered by the user in the login screen.

Another issue this projects aims to contribute is in efficient diagnosis of design properties. Currently, designs are not standardized or computerized to enable such properties analysis – only by hand which is lengthy and error-prone. A result of the definition of a standard language for describing the design is that algorithms can be written to interpret and analyze it. It is our aim to show that by writing these algorithms designs can become more robust and contribute to a better Web Application products and more efficient and cost-effective development cycle.

Thesis Organization

This thesis consists of several chapters. Following is a summary of the chapters in the order in which they appear in the thesis:

Introduction

We introduced the state-of-the-art and problems that were identified. We state the objectives and theoretical background. We

introduce Navigational Design Patterns and the intended contribution.

In **Setting the Scene** we position this process within the Complete Designing Process. We describe Functional Requirements and following sub-processes, such as: how to express design, re-usability and maintenance and properties diagnosis.

In **Navigational Design Patterns** we explain what Are Navigational Design Patterns. We provide a set of examples and existing problems (e.g. defining, searching).

In the chapter on the **Navigation Design Pattern Language (NPDL)** we introduce the concept of the language. This is followed by a description of the elements and construction; basic building blocks, combination (exceptions) instantiation of Design Patterns and analysis. We formalize the addition of the following extensions: authorization and exceptions.

In the chapter **Case Study: CampusNet** we define the patterns for a university Web application and design a part of the real CampusNet system using NPDL. We diagnose the design and compare the results to the actual system. We introduce the ability to connect the design to a code generator to come up with a mock-up system that can actually be used.

In **A Visual Tool for Designers** we describe an option of an implementation using the Client/server approach to create a visual tool for NPDL designers.

In the **conclusion** we summarize the thesis, discuss the results and provide a list of future work that can be carried out as a

continuation to this project.

Bibliography

Appendix A: code samples

Chapter 2 - Setting the Scene

The subject of this thesis is high Level Web Applications Navigational design and diagnosis – a compact and standard language for definition. While this statement is concise, there is still a need to define some terms and concepts before the specific thesis can be introduced. Justification of the problems that the thesis solves is served by clarifying our views on the underlying processes.

There are some well-established notions that need to be presented because they are pivotal within the domain of navigational design. As always, formalization presents the need to make some assumptions about the domain, so those assumptions need to be presented, as well.

In order to clarify our view of the domain, we informally describe the main concepts and assumptions on the domain:

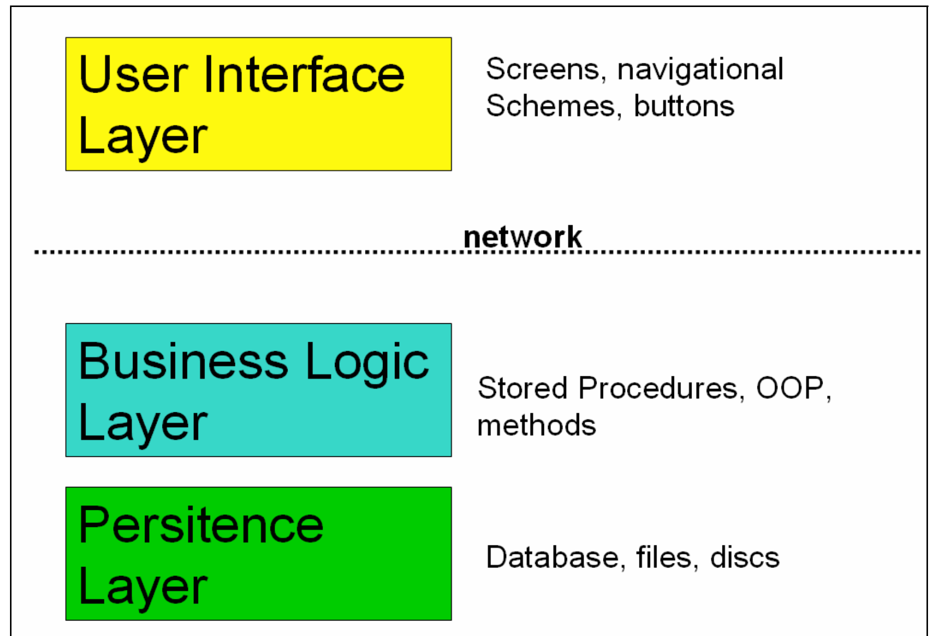
- Web Applications are software products (we focus on Information Systems, such as CRM, Portals or Project Management) based on Web architecture, mainly having a browser as a mean of presenting the user interface for the users
- Navigational Design is the mapping of some of the Functional Requirements to a concrete set of screens and
- Web Applications Navigational design is a definition of the separate categorized modules (such as course administration and activity management) the navigational connection inside and between these modules.

- Navigational Design Patterns are *abstractions*. We elaborate and show how they are currently defined in text or graphically.
- Our work is inspired by theoretical efforts to define and categorize navigational design patterns by people such as: Germán, Schwabe, Rossi and van Weile. Their work is presented in publications such as [GC1], [RSL3] and [WV1].

Position within the Software Design Process

The software development process in general and specifically Web Applications (WA) design is a complex process. Mainly one can divide it into three parallel layers: the persistence layer design (e.g. Files or database), the business logic layer (e.g. Objects, procedures/functions) and the User Interface layer (e.g. Screens, navigational schemes). The systems designers have the skills of mapping the single set of functional requirements into these segments and then design the solutions for each separate set of requirements. Our thesis relates to the design of the navigational scheme within the user interface layer. The knowledge and skills of the navigational designers differ from those of business-logic and persistence designers.

It is assumed that the information for the designers of the navigation has been provided by the designers of the business logic and persistence layers, as a correct process dictates. This is because the User Interface design is affected by decisions made by the business-logic and Persistence layer designers, rather than the other way around.



Development Process Layers

The positioning is important since it demonstrates that the skills required by designer of this layer differ greatly from those that are needed for being a business-logic or persistence designer. The designers of the navigational layer have an understanding of the way human users interact with an application of the specific sort they are designing, but also general rules that apply to HCI. In accordance with the ability to transform abstract requirements into solutions and experience, they initially create a high level design of the application at hand. It is therefore in this realm that we aim to postulate the thesis and the benefits of its applicability.

Functional Requirements

Functional requirements (FR) are the means by which the intended users of the application describe their expectations as to how the system must operate and how they should use it to achieve the process that the system automates or supports. Functional

requirements come in many different shapes and forms. A standard that is widely adhered to has not been established yet. However it is visibly a textual document that is separated from the following stages of design and implementation.

Interface Requirements
This section will enumerate Interface requirements, those are requirements that affect/interact with systems external to this system, for example "For each new employee in the ABC system a new record will be created in the XYZ system".

Example of Interface Requirements table:

Req.#	Requirement	Description	External System
20	The system shall record an acknowledgment when a new employee is created into the XYZ table	The XYZ system uses these records to update the time stamp of an employee record.	XYZ - security system
2	The system will mail the month-end report message to the "Director" at each month-end.	This E-mail is used to manually load data into the Director system.	Director system

Example of functional Interface Requirements
(http://strategis.ic.gc.ca/epic/internet/ininfodev.nsf/en/dv00196e.html)

One of the tasks of the navigational designer is to extract the portion of the abstract functional requirements, which are destined to be implemented in the user interface layer – i.e. as screens and components that the human users interact with. This process is not in the scope if this thesis, but the output from it provides the starting point for the designer with the tools we intend to provide. Moreover, these subsets of requirements that have been identified become a part of the design definition and diagnosis of the design, as suggested by our thesis results.

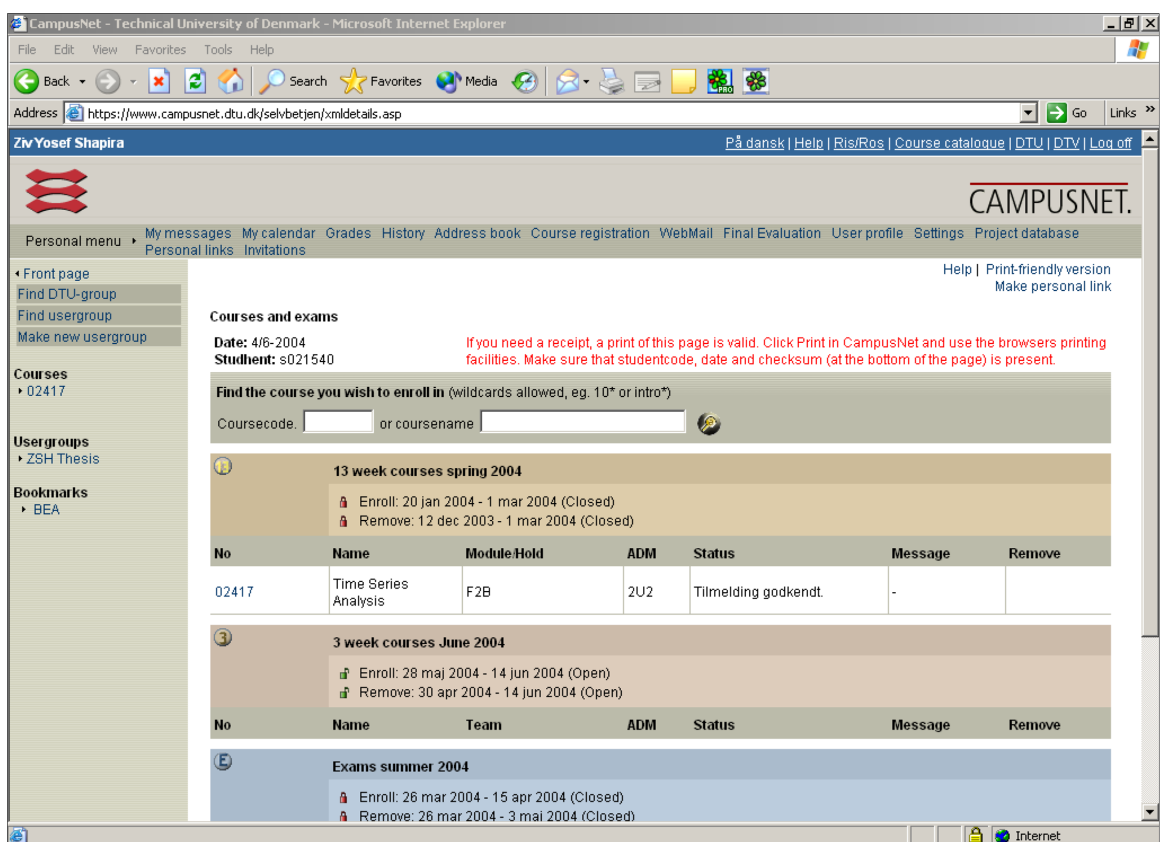
We have not seen so far a convincing and formal way of connecting the FR to the design in such a way that there is also a

reverse process of conveying why a specific navigational design has been chosen with respect to the given FR. Notably, little work has been done regarding the derivation of architectural descriptions from functional requirements specification [AVL1].

Elements of Navigational Design

The elements that are the "alphabet" of the navigational designer are constantly explored and refined. The main reason for this is the introduction of new technologies for application development, web applications being one of the more recent and the focus of this thesis. Within this scope elements that the designer uses include:

- Screen elements – a set of visual elements that enable users to interact with them. Examples include: buttons, links and text boxes.
- Screens – sets of elements that provide a set of functionality within a given context. Examples include: updating course information or entering a daily appointment in a calendar. Screens are viewed by Web browsers.
- Connections – paths betweens screens that can be achieved by hypertext links or by buttons for example. The set of connections indicate all the possible paths a user can make while using the application.
- Authorizations – a set of users or users groups (e.g. Administrators) that are allowed or are not allowed using a screen element or entering a screen. Authorization differentiate between different type of persons within the organization that uses an application with respect to what modules (set of screens) or functions they may use or what information they may view.

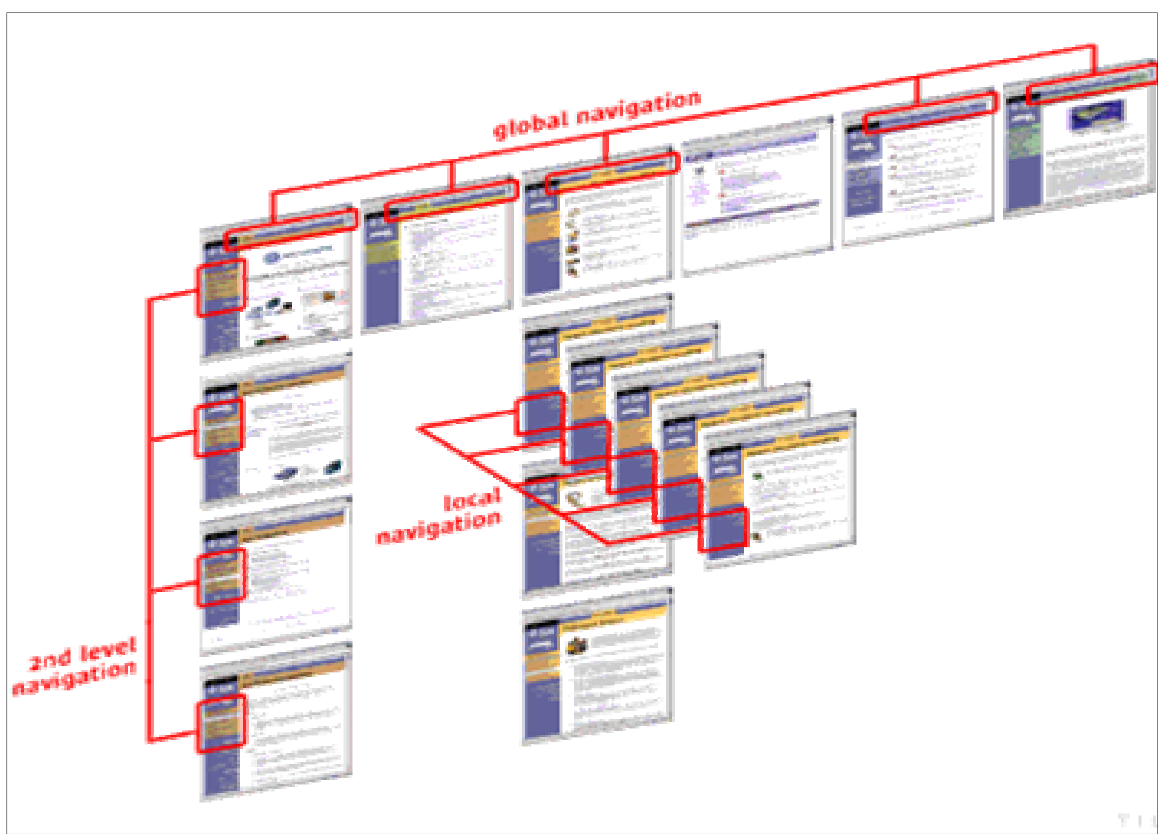


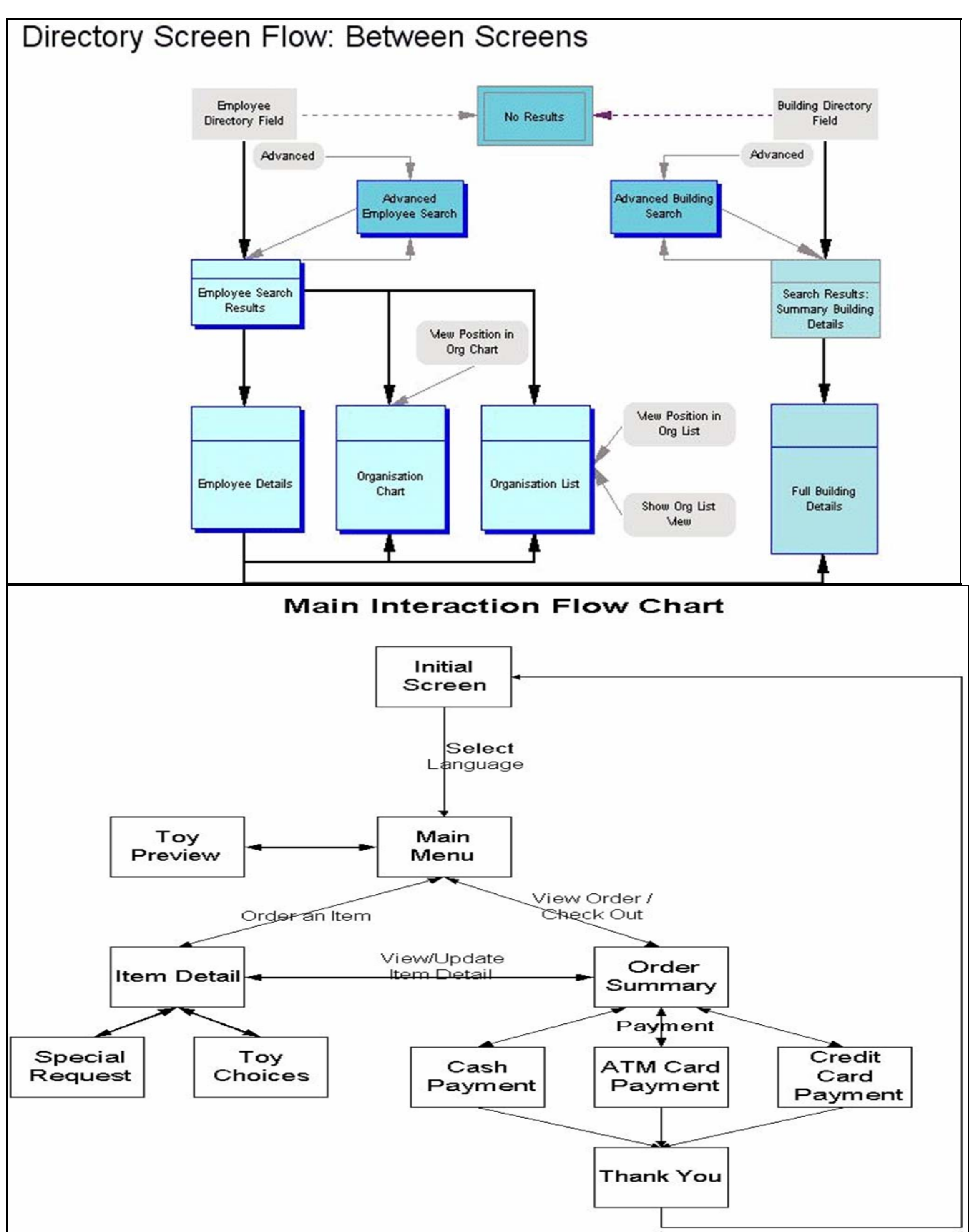
Example of a screen in a Web Application (CampusNet at DTU)

Expressing the Design

There are obviously many different ways in which the navigational designer can express the design. These means vary from being descriptive (text based) to graphical, as seen in the image below, or some combination of the two. The importance of a concise and clear design is that is the basis for the successful continuation of the development process – i.e. Development and maintenance of the application. Evidently both of these methods are not successful in terms of being compact or even readable when the application exceeds even a small size, as most often happens. Some modeling approaches have been discussed in [**KRS1**].

The thesis states that such a method is to be found where the description is both standard and compact so it is clearly understandable and manageable by other as well as the designer, who produced it.





Examples of various representations of navigational design

Reusability and Maintenance

Reusability is a very important aspect in software design. It states that existing solutions that have been produced may be easily applied to new problems in such a way that the solution need not change its basic structure. Design patterns have been identified as means of expressing solutions to problems faced by designers in a way that is reusable. Researchers such as Schwabe and Rossi "...introduced navigation patterns as a way to record, convey and reuse design experiences. Navigation patterns [...] show how to go beyond the basic Web navigation paradigm to solve recurrent problems." [SREL1]. A need for reuse is further discussed in [NNK1]. Another aspect of reuse is that of the patterns themselves, i.e. that more basic pattern can be reused when defining more complex patterns that share similar features. For example, a catalog pattern and a search pattern should be easily reused (after being designed), when creating a pattern for a catalog with a search function. Our findings show that although patterns have been defined, they are not easily searchable or matched to problems that designers face due to the way in which they are presented. An additional set of reasons for this problem is mention in [**GC1**], e.g. Bad Naming of patterns, no catalog for patterns, several patterns that solve similar problems and lack of classification of patterns.

Another important, and somewhat related, issue is that of maintenance. In software development, including Web Applications, the design is constantly tested, changed and updated due to requirements changes, technological advances and competitive constraints. The process of keeping the actual design up-to-date with those changes that will be incorporated into the software is called maintenance. The lack of standardization when describing the navigational design of WA makes it extremely hard

to maintain with respect to the aspects mentioned above. The reasons for this are the size of a final design in terms of text and/or graphics and the inability to process it mechanically.

Diagnosis

An important aspect, which relates to design as much as it relates to the other phases of software development, is the ability to perform diagnosis of the design in order to verify or investigate properties, such as reachability, authorization breaches and "bottlenecks". For a concrete example, the designer who has completed the design would like to know which screen is accessible to most user groups (and thus to most actual users) and relates to most of the functional requirements. Such a screen is likely to be used very frequently as it is relevant to most users in most of their interactions with the system. This makes the need to either indicate to the developers to provide an efficient implementation of the screen or lead the designer to consider splitting the functionality of the screen to two or more smaller screens. In practice, we found it very complicated to diagnose designs in their current format or to test or verify how maintenance operations affect the existing design in terms of the original or new requirements.

Manual diagnosis is not longer practical due to the complexity of recent and emerging applications. The complexities of designs required to meet present needs are simply too great. There is a clear need for automation of this diagnosis and for an ability to create this automation based on standardizations of designs rather than tailoring algorithms for each design after it is created.

We aim to show how our theory and solution partially or fully

address all these issues.

Chapter 3 - Navigational Design Patterns

This chapter introduces navigational design patterns – the concept and examples. The idea of patterns has originated in the work of Alexander [AIS1] and comes from architecture. The adaptation of the patterns concept to SW design has been initiated by E. Gamma in [GHJV]. Patterns are means of conveying abstractions of best-practice design solution to reoccurring problems. The abstraction ensures that the solution is general enough to fit many cases (instantiations) of the problem. The best-practice part ensures that the solution behind the pattern is well-tested and has been found suitable so it does not impair the robustness of the entire design in which it is used.

The main power behind the patterns is not in the innovation of the solution that they represent, but in the way that this solution and its associated attributes are conveyed to designers and developers alike. Beyond unambiguously clarifying the designer's intent, patterns are meant to be a reusable representation of the solution so that they may be reused within the same overall design for several of the problems. An experienced designer may already encounter the solution, but beginners will find it very useful both for concrete design and as learning tools for future designs. An important part of this ability to represent solutions is in the fact that patterns can represent solutions to small part of the application (such as search functionality), entire modules (such as course registration) and even to complete applications (e.g. E-commerce or University Portal, which may even have several patterns). This property is what makes patterns modular in themselves, but because of their

standard format enables combining the patterns of small solution into those of larger ones, thus inheriting their properties and qualities.

Navigational Design Patterns are Patterns that are used to design the user-interface layer of Web Applications, i.e. the screens, their structure and the navigation between them. The field of patterns for this purpose is relatively new. Navigational Design Patterns, including ones for Web Applications, have been discussed in research, such as [MB1] and [RSL1], and have provided inspiration for the foundation of his project.

This chapter discusses Navigational Design Patterns in an attempt to inspire a standard way of describing all existing and future navigation patterns. The patterns are shown to be a sound basis for navigational design as was introduced and elements that require a formal language for concise and unambiguous description, one that will be introduced in the next chapter.

Pattern Types

In order to enable an effective mechanism for cataloging and searching design patterns it is vital to classify the known patterns using a small set of types. At this stage, the following types of patterns were identified:

- **Architectural Patterns**
  Design patterns that solve problems related to the design of the overall application structure. (e.g. Cycle, wizard)

- **Process Patterns**
  Design patterns that solve problems related to the way user execute business processes via the Web application. (e.g.

Shopping cart, login, calendar)

- **Presentation Patterns**
  Design patterns that solve problems related to the way visual components, content and data are presented to the application user. (e.g. Virtual product, News)

- **Usability Patterns**
  Design patterns that solve problems related to the way users interact with the Web application's visual components. (e.g. Group location awareness, landmarks)

One example of a navigational design pattern is a collector. This pattern is a process type pattern, which provides a design solution to a process in which users need to collect a set of items (usually of the same type), in order to perform some operation on the set at a later stage. A widely used implementation of this pattern are shopping carts in e-commerce sites, where users collects items (books, flowers) and later want to pay for all of them once. Another example, in a university application, would be students that collect courses for a given semester and wish to register to them as a set after the courses have been selected.

This thesis focuses on the first three types. The reason is that they represent the types of solutions that are of interest to us in robust navigational design and the properties that we wish to diagnose in this designs. Usability patterns are more the focus of web Designers and are considered a refinement, which is discussed as future work. We indicate however that the extension to use these patterns in the overall scheme is clearly possible and integrated.

Elements of Navigational Design Patterns

The definition of what are essential attributes for patterns has been discussed in details in [KUH1]. NPDL incorporates a set of common attributes that all patterns share. These attributes create a common 'interface' for all the patterns, so that they may be analyzed as patterns and as instantiations (specific designs). The pattern architect provides values for the following set of attributes in a pattern:

Attribute	Meaning
Pattern name	Conveys the essence of the pattern
Intent	What is the intent behind this pattern? What problem does it solve?
Also known as	Other known names for this pattern (e.g. Literature, common references)
Motivation	A scenario illustrating a typical problem and the solution using the pattern
Applicability	In what situations does the pattern apply
consequences	How are the objectives supported? Trade-offs of using this pattern. What are the results?
Known uses	Examples of applications, sites that use the pattern Links
Related patterns	Patterns that are closely related to this pattern
Screens	The *screens* that make up the navigational pattern. Can also be inherited from more basic patterns.

The designer of the application, who instantiates the pattern, provides the second set of values for the following common

attributes:

Attribute	Meaning
Module name	Name of the module that will be implemented using the pattern
Fulfilled Requirements	The set of functional requirements that are fulfilled by the module
Reachable screens	Screens in other modules that are reachable from this module

Once all attributes have been given values, the pattern is instantiated to represent a specific module in the application. The instantiation encapsulates both the functional requirements and a best-practice solution (set of screens and navigations) to the requirements.

Screens

The screens are part of the attributes of a pattern. A definition of a screen in NPDL consists of the following attributes:

Attribute	Meaning
Identifier	Unique identifier of the screen. Screens that belong to the shared library of patterns have a unique identifier in the entire library.
Name	Name of the screen
Elements	A list of *screen elements* that make up the screen. See the attributes of screen-elements.

Attribute	Meaning
Module name	The module name (supplied by designer). A screen identifier in a module with the module name is unique in the application design.

Screen Elements

Following is the set of symbols in NPDL and their meaning:

Symbol	Meaning	Required Information for Design
↔	Horizontal bar (menu)	• ID • name
→	link	• Type (link) • Name (descriptive for dynamic links and fixed for constant links) • target (screen ID) • Does open in a new window?
■	button	• Type (button) • Name • Related function • Is clickable on entry?
%	manual input - any text - case sensitive - only numeric - only text - date - URL/path (hyperlink)	• Type (inputbox) • Name • Input type (cf. meaning) • Is input masked?

Symbol	Meaning	Required Information for Design
╧	single selection from multiple choices	• Type (select single option) • name • data source
╨	multiple selection from multiple choices	• Type (select multiple option) • name • data source
▒	Dynamic Content	• Type (dynamic content) • data source
≈	Check box (2 states)	• Type (checkbox) • Name • Related function • Is checked on entry?
█	Data table	• Type (data table) • Name • Data source
%RO	Display text	• Type (display text) • Name • Data source

The other part of the design is how to connect the separate patterns to reflect the whole design. For this a set of connector type is required:

Connectors (navigational paths)

Meaning	Required Information for Design
One exit possibility (e.g. Exactly one button/link leads from page A to the destination page B)	• ID • source • destination • parameters list
One or more exit possibilities (e.g. More than one button/link leads from page A to the destination page B)	• ID • source • destination • parameters list

Exceptions

Navigational design patterns convey the human interaction with the Web application. Normally, the patterns refer to successful interactions. However, in practice there could be failures within the interaction process that relate both to the Web infrastructure (e.g. HTTP) or the logic of the application (e.g. by trying to register to a non-existing course or adding an activity in an invalid date). The reasons they are required to be expressed in the design stage are:

- The designer is the responsible person for addressing these exceptions
- The skilled designer knows how best to address them in a consistent manner with the entire design
- Well-tested treatment of exceptions should become part of patterns on the abstract level
- Treatment of exceptions can themselves be patterns and thus can be expressed using **NPDL**. Common exception solution to several patterns can then be reused.

In addressing exception it is vital that the designer is able to express the following points regarding the exception:

- What is the trigger for the exception? For example, failure in the operation or invalid data-entry format.
- Description of the output produced when the exception is encountered. This ideally will be in **NPDL**, or reference a known pattern for treating such an exception, if such a pattern exists.

These exceptions in the normal operational mode can be expressed using **NPDL**. The proposed language has given this option for the designer for the above mentioned reason and also for being able to later explore properties of specific designs. The explored properties section presents them in details. Allowing the pattern designer and their users to express and address exceptions at the design level makes the design more robust and less error-prone, because the designer is forced to address issues which inevitably will arise, but should be defined by the designer and not during implementation.

Authorizations

Role-based grouping of users is a popular authorization scheme by which access rights are enforced in Web applications. Access rights are defined in order to achieve the following goals:

- determining which data can be viewed by a user
- determining which functions can be activated by a user
- determining which screens can be viewed by a user
- determining the navigational paths that a user can follow when using the application

Access rights can be related to the application on several levels:

- Components in screen. Parts that display data or that activates a

function (e.g. link, button, data view)

- Complete screen or set of screens. Entire screens that a user is allowed or not allowed to access completely (e.g. Financial module, project management module)
- Navigational paths. Transitions between modules/sub-modules in the application that the user may or may not explore (e.g. Access financial records of a customer, but can view contact details).

One way of defining authorization schemes is by creating roles (logical groups), based on the types of users of the application (e.g. Employee, manager, super user, student). Thereafter, actual users are assigned to one or more of these roles and are associated with them from the moment they log in to the system. **NPDL** utilizes this approach and enables the assignment of groups of users to each component, screen and navigational path, so that some properties of authorization may be explored at the high-level design stage. This association enables automatic generation of code and enables simulation of various user types' usage scenarios.

A pragmatic approach has been taken in the design of NPDL in order to enable both flexibility and compactness of the access rights association. User groups (representing various authorization schemes) can be individually included or excluded from the right to use a components, screen or navigational path. In addition the entire set of groups may be included or excluded in the access rights. Having this option enables the designer to specify which groups are allowed and which are not, without having to make global assumptions, such as: "if no access rights are specified for a given function, then all groups are allowed to use that function".

Such global assumptions, where access rights are not specified,

cause undesirable symptoms in the design process. The following two use cases demonstrate some of these symptoms:

- errors

In this use case a function F is accessible for all user groups except group A. If the designer neglects to reject group A from using the function F, due to the global assumption, the implementer allows group A to access the function F. This access causes a security breach. If, on the other hand, the designer must specify all the authorizations, the margin of error decreases.

- Maintenance

In this use case a function is allowed for all users in group A but not for the other 10 groups. The designer is forced to specify 10 groups as rejects. Using NPDL it is simple to reject all groups or allow group A.

Once the authorization scheme has been specified by the designer, it is possible to investigate properties related to it. These properties are important for the correct function of the system and can save implementation and maintenance resources, if detected at the early high-level design stage.

Symbol	**Meaning**	**Required Information for Design**
+*	All groups allowed	• 10000
-*	No group allowed	• 10001
$+a_1$	Users from authorization group a_1 allowed	• ID

Symbol	Meaning	Required Information for Design
$-a_1$	Users from authorization group a_1 not allowed	• ID
∩	And	
U	Or	

Basic Navigational Design Patterns

Basic navigational design patterns are patterns that cannot be broken down into smaller parts. Since the patterns existence is based on it being a best-practice solution to a problem, each pattern that solves a basic problem (single requirements, like searching, displaying or collecting) cannot be reduced, since it will lack the elements for a solution.

The best way to illustrate Navigational Design Patterns as they are today is by giving some examples. These examples are inspired by descriptions in [RSL1] and [WV1] as well as original suggestions by us. We have tried to illustrate patterns of all types, but keeping focus on the types of main interest to the thesis.

1. Basket

Description: a collection of items and operations on the collection of items.

Context: A need to enable users to keep track of items selections during navigation, making them persistent for future processing when user decides to.

Goal: Decouple product selection and processing.

Type: Process

Example: DTU course catalog (university)

Number of Required screens: 3

Screen	Description
Items overview select action	What items are in the basket? What operations can be performed on the items?
Action parameters	Parameters for the operation, such as printer name (for PRINT)
Feedback	Result of applying the operation on each item

2. Calendar

Description: a collection of activities separated by periods, e.g. Days, weeks and months.

Context: A need to enable users to follow and manage basic calendar activities.

Goal: Efficient management of activities based on a periodic scheme.

Type: Process

Example: My Calendar Module in CamusNet

Number of Required screens: 5

Screen	Description
Monthly View	Monthly calendar view
Weekly View	Weekly Calendar view
Daily View	Daily calendar view
View appointment	Information on a specific appointment without ability to update
Enter Appointment	Enter information on a specific appointment

3. **login**

Description: a personal and secure method for reaching the application functionality.

Context: A need to enable users to securely reach the application functionality. Personal entry allows customization, authentication and role-based authorization.

Goal: Secure and Role-based entry to system functionality

Type: Process

Example: Campusnet login

Number of Required screens: 2

Screen	Description

Screen	Description
Login	
Main Screen	

4. Address Book

Description: a collection of contacts that are relevant to the application. Includes contact name and details.

Context: A need to enable users to manage contacts for personal and business purposes.

Goal: Efficient follow-up and management of contacts

Type: Process

Example: Address Book in Campusnet

Number of Required screens: 3

Screen	Description
Contact list	
View contact details	
Insert/Edit contact details	

5. Catalog

Description: a collection of items that can be viewed in categories or individually.

Context: A need to enable users to view items and insert and edit a those items.

Goal: Display items to intended audiences for information or marketing purposes, for example.

Type: Presentation

Example: Participants in Course

Number of Required screens: 2

Screen	Description
Items Catalog	
View Item Details	
Edit Item Details	

6. University Portal

Description: an application that allows staff and students to manage information regarding the academic life at the university, such as courses, grades, messages and activities.

Context: A need to manage information regarding academic activities within a higher-education institute such as universities

and colleges.

Goal: Provide a robust and comprehensive design of an application of this type.

Type: Architectural

Example: CampusNet

Number of Required screens: 50. This is an example; actually there could be several patterns for this type of application with a different number of screens.

7. Group Location Awareness

Description: a hierarchal set of links representing the navigation path from the current screen to the main screen.

Context: A need to enable users to associate their current location (screen) to the structure of the application.

Goal: Provide a permanent reference about the user's location in the hypermedia space.

Example: breadcrumbs

Type: Usability

Number of Required screens: 0

The additional benefit of navigational design patterns and a concise language that describes them is the ability to combine patterns, as discussed also in [WV1] and [MB1]. A patterns that is designed as a combination of more basic patterns is of course a pattern as well, but the fact that it is a combination of navigational design patterns gives some important and useful attributes to a language that easily enables this combinations to be defined. First, the language has inheritances properties, meaning that the complex pattern acquires, by the way that it is defined, the properties and attributes of the basic patterns, such as the screens and navigational paths. When they are updated, the result propagates through the inheritance chain to all the complex patterns. Second, the need and ability to combine patterns enhances the reasoning behind the standardization of a pattern definition (language). Only by adhering to a standard does the combination become possible, since the complex pattern designer can integrate the basic patterns in the same way. Furthermore, the resulting pattern can be combined as a part of another pattern using the same standard. Another advantage is that the complex pattern is more robust if built using patterns that are themselves well-designed solutions to the sub-problems. This ensures that patterns scale well, but still maintain robustness. This relationship between patterns creates and **hierarchal connection** between patterns.

An example of a complex pattern is a catalog with search. The pattern described below enables users to use the catalog pattern combined with the search pattern capabilities.

1. Catalog with search

Description: a collection of items that can be viewed in categories

or individually and be searched by some criteria.

Context: A need to enable users to view, insert and edit them as well as search for subsets of the catalog.

Goal: Display items to intended audiences for information or marketing purposes, for example. Allow potential users to search for items that match their current needs.

Type: Presentation

Example: Students searching for courses to register to

Number of Required screens: 5

Screen	Description
Items Catalog	A list of the items in the catalog
View Item Details	The details of a specific item in the catalog
Edit Item Details	Change and update details for one item
Search criteria	A criteria for searching items, like color or price
Search results	The result of the search

Another example of a combined navigational design pattern is a basket with login:

2. Basket with login

Description: a collection of items and operations on the set of items that is specific to each user of the application.

Context: A need to enable users to keep track of personal items selections during navigation, making them persistent for future processing when user decides to.

Goal: Decouple product selection and processing for each individual user

Type: Process

Example: Amazon.com (e-commerce)

Number of Required screens: 4

Screen	Description
Login	Log into the application using a personal user Identifier and password
Items overview select action	View the items Set the action to be performed
Action parameters	What parameters are needed, for example a printer name
Feedback	What is the result of the action

The hierarchical operator ***h*** returns a new pattern based on the definition of the basic patterns that are provided as input. The connectivity of the screen within the pattern is the provided by the pattern's designer. For example, from which catalog screen can the search criteria screen be reached and to which screens does the search results screen lead.

Summary and Discussion

At this stage we have attempted to layout the incentive behind the thesis – the navigational design patterns and why a standard language for defining and designing Web applications using them is necessary. We aimed at pointing out why a design based on navigational design patterns is a more robust yet flexible design that is more easily implemented and maintained rather than a design "from scratch" where the solution are based on no or poor experience and on "re-inventing the wheel" approach (although maybe unintentionally due to lack of experience). It's our claim that a language that enables the definition and use of patterns in a compact, standard and maintainable way will accelerate the acceptance and use of the patterns, leading to better navigational designs. We have given an example of CampusNet, the DTU portal as an example of a Web application and identified the patterns it has.

The way we suggest to combine patterns is summarized and expanded. It is clear from our choices that we have a vision of a hierarchical structure of navigational design patterns. We find this structure both meaningful enough to display the connections between patterns and flexible enough to display the types of connections. In our presentation we have shown a simple connection by combining patterns, as a type of inclusion connection. There are however more possibilities, such as **conditional combinations**. In this case, the transition between patterns is based on some action the user performs and can therefore results in different navigational paths. One example would be an instantiation of a **persistent catalog** in an e-commerce application. The designer wants to keep users who wish to exit a product category interested, so they will be diverted to a collector of a related item. Thus the complex **persistent catalog** (solves this

need) is a conditional combination of a **catalog**, such that if the user want to purchase the items the combination is with a **collector**, otherwise the combination is with another **catalog,** depending on the user's action. Another example are **while combinations**. In this case, the transition between patterns exists as long as some condition is satisfied. As an example, we define the complex pattern **Collector with view** as a while combination of a collector and a non-empty collector state. The combined pattern is a **virtual product display** pattern. The resulting pattern solves the problem of the user's need to view the items in the collector successively before performing an operation, such as purchase or register.

The next chapter deals with the language we propose and that demonstrate the above-mentioned points in practice.

Chapter 4 - Navigational Patterns Definition Language

This chapter introduces the proposed Navigational Patterns Definition Language in details. The **Navigational Patterns Definition Language (NPDL)** is the result of the research in this thesis project. It provides a tool for navigational designers to express their design in a compact, understandable, standard and maintainable way. The need for such a language and its properties has been discussed, e.g. in [WV1] and [MHG1]. The purpose of the language is to provide a unified solution to all the participants in the process of Web applications user interface (UI) design. The contribution described herein is with the definition of the language with a perspective of all the issues that have been identified and discussed in previous chapters. In the next chapter we will describe a concrete example of design using NPDL.

Motivation

We start by explaining the motivation behind the definition of a language for defining navigational design patterns and for describing specific navigational designs using patterns. The motivations stems from the problems we have identified in the current way that navigational design patterns are defined and used within designs. The first problems we describe are that there is a **no standard** for the attributes of an abstract design pattern. Several researches (e.g. [NNK1], [RSL3] and [SSBZ1]) have

attempted to provide a set of attributes, but they differ and have not converged to a single agreed standard. It is our belief that this is due to the fact that these attempts have only taken into account a subset of the requirements that these attributes need to serve. Another problem we have identified is the way in which patterns that have been defined are presented to designers. The **presentation scheme** always consists of textual explanation and graphics of the navigational scheme suggested by the pattern or an image portraying the use of the pattern in an application. Research and sites regarding this topic reflect this problem which worsens as the application's navigational designs get more complex. Another problem we found is the **inability to effectively search** a repository of navigational patterns due the lack of standardization, language barriers and presentation schemes which are not effectively searchable (e.g. graphical representations). Some suggestions for a patterns catalog have been suggested (e.g. [**GC1**]), but without the standardization we claim they would be ineffective. A major problem resulting from the above is that there is no standard means of **describing the navigational design** of web applications with patterns, even if they are found and chosen. This is because the professional languages of the pattern designers and navigational pattern designers are completely separated, although in practice they are persons with identical background and a part of the same process, i.e. The pattern designers output is an input for the Web application designer.

All these problems have prompted the research behind this project. The result is described in this chapter – the navigational design patterns definition language.

Concepts

The concept behind NPDL is to use the standardization of navigational design patterns we propose and the power of Functional Programming as a common tool for the roles involved in this stage of the software development.

The language enables Navigational Design Patterns **architects** to do the following:

- Define Patterns in a standardized way. The patterns have a set of common attributes and unique attributes
- Create shared libraries of these patterns
- Compose complex patterns from more basic patterns

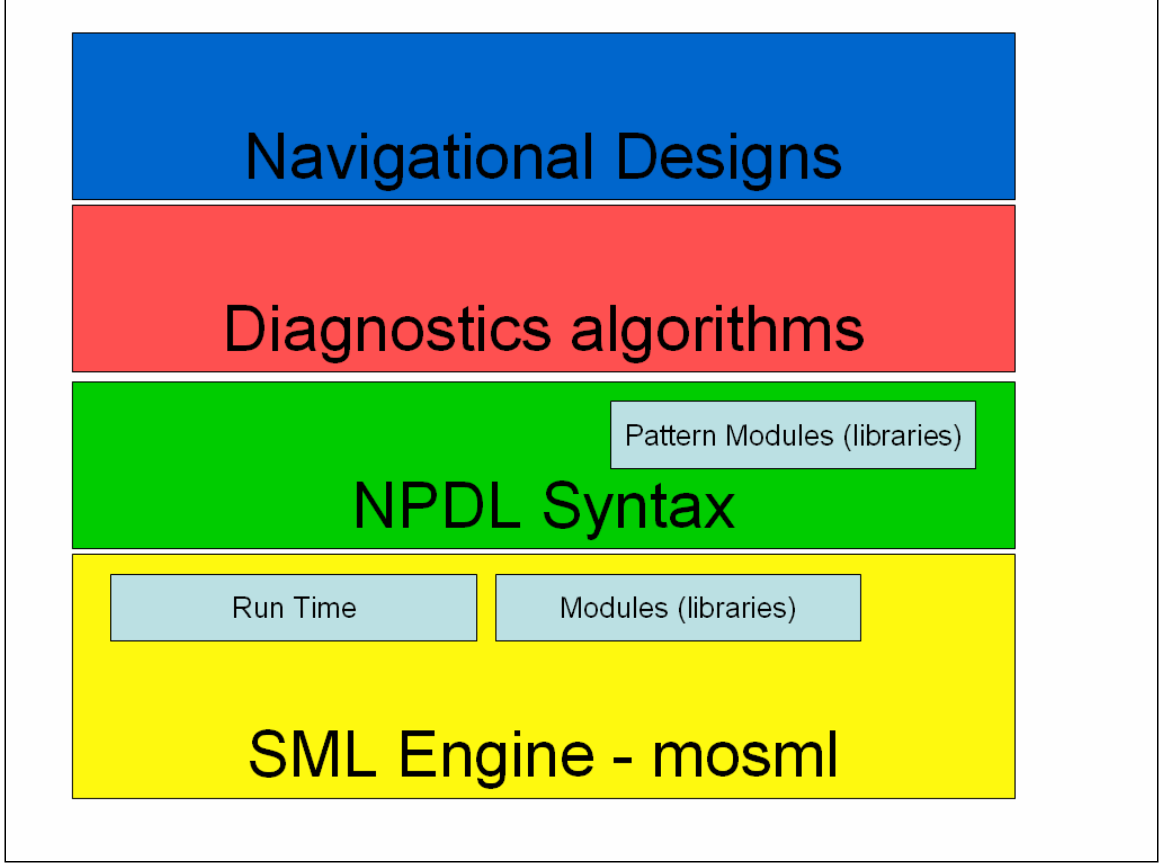
It also enables a Web Application User-Interface **designer** to achieve the following goals:

- Convey the design of the presentation layer: screens, their components and the navigational scheme in a standard and compact manner
- Relate the design to the software requirements specification
- Use a library to search and instantiate navigational patterns for specific modules. Furthermore, relate the solution, which is purposed by the pattern, to problems that the requirements pose for each module
- Produce coherent and updated documentation of the proposed design

Finally, it provides a solution for **Quality Assurance** (QA) **and Integrators** who have the following tasks to perform:

- Diagnose the design properties (e.g. recurring screens during navigation, authorization conflicts)
- Validate the cover of functional requirements for the UI by the proposed design

- Create search-mechanism on the patterns catalog
- Transform the output of the design to various formats (e.g. XML, HTML, graphical representation)



Functional Layers in NPDL design and use

NPDL provides an abstraction of the required components, screens and navigational paths regardless of the underlying technology.

Functional Programming - SML

We would like to first explain why we have chosen an existing programming language as a basis for NPDL. This choice has been made after we have decided on the properties of the language. It is

natural to create a unique syntax, but it has been a choice to explore whether an existing language has most of all the basic requirements and can therefore be adapted to the specific needs. The basis for NPDL is the functional Programming Language called SML (Standard ML). The language specification and concepts is described in [**HAN**]. SML is a part of the functional programming languages (FPL) family, which also includes: Erlang, Haskell and Scheme.

One main reason for selecting SML is that it is widely used and therefore can be considered a **standard** in it's domain and has a predicted short learning curve (since we use only a part of it). Although naturally navigational designers do not currently use SML for their design (this is one of the innovations and contributions), we predict that the syntax we use can easily be learned, even by novice designers or programmers. Another main reason for selecting an FPL as a platform for NPDL is that it enables the designer to **compactly define** what the goal behind a statement is rather than having to use an elaborate syntax to describe how to do it. The latter is a signature of OO and procedural languages, like Java or C. the syntax of SML is successful at this and is a main part of the requirements we have stated for NPDL. Another powerful feature that we have harnessed for NPDL is that the **interpreter** of an SML code analyzed and displays the results in an understandable format, so that the any statement in NPDL can be further analyzed for properties, such as **type checking** (e.g. Of parameter types) and **hierarchy expansion** (e.g. Of hierarchical structures). These properties belong to the navigational design and though the design itself is compact, these properties can easily be accessed by other designers or by the developers. Furthermore, the use of SML ensures that code that is used for **diagnosis** of the designs can be written in SML as well,

and therefore be clear and written by the same persons as the patterns designers and the Web applications designers (same SML skills). The results of the diagnosis are provided by the SML interpreter and therefore can be comprehended by all parties involved. Some of the properties that can be analyzed on an existing NPDL design will be discussed later in this chapter.

We have however defined syntax for NPDL using SML features. The features that have been used are described below:

- *Structure* – a sequence of declarations, such as types, constants, variables and functions
- *Signature* – a sequence of specifications. Includes all information that needs to be known about a module, but nothing else
- *Functor* – a function from a structure to a structure. Used as an abstraction mechanism

Functional Requirements

The first elements of NPDL we introduce are the functional requirements (FR). The signature of functional requirements in NPDL reflects the set of attributes we defined for them in previous chapters. This signature is introduced below:

```
signature FunctionalRequirementSig =
sig
   eqtype Id
   type Description
   type Requirement
   val reqDesc: Requirement -> Description
end
```

Functional Requirements Signature

The NPDL library contains a default implementation of the FR

signature as shown below:

```
structure defaultRequirement : FunctionalRequirementSig =
struct
   type Id = int
   type Description = string
   type Requirement = (Id * Description)
   fun reqDesc (id, desc) = desc
end
```

Default Functional Requirements Implementation (NPDL library)

The difference between this default implementation and other is in the type of the attributes (and not in the attributes themselves). In the default implementation the FR are identified by a unique integer Identifier, the Description is a String. A navigational designer can use the signature to define another implementation and use it instead in the NPDL design that follows. We have introduced the default implementation as we believe it complies with almost any FR description (i.e. a numerical identifier for a requirement and a textual description).

The set of functional requirements is provided in the NPDL library and the code is:

```
structure FunctionalRequirements = SetFct(type element = defaultRequirement.Requirement)
```

A set of default Functional Requirements (NPDL Library)

The navigational designer will enter the FR of the specific application (e.g. CampusNet) using this signature for each of the requirements, when designing in NPDL. An excerpt of code for entering requirements is shown below. Here 2 requirements are being inserted into *fr,* a set of default type functional requirements:

```
val fr = FunctionalRequirements.insert((1,"register to courses"),
          FunctionalRequirements.insert((2,"view student grades")
                        .....
```

Example of defining specific Functional Requirements

We will later demonstrate how this FR are integrated into the designed modules, so that there can be a diagnosis of which modules implements which Functional requirements. Alternatively, a reverse diagnosis can reveal which FR are used and which are not used by analyzing the designed modules. Both of these properties are pivotal in the software engineering process and will be discussed in the diagnostic properties section.

Pattern Types

We have mentioned four types of navigational design patterns. In NPDL we handle three of them: architectural, process and presentation patterns. Usability patterns are not handled by our proposed language because they are different in the way that they are described and relate to lower-level navigational design, which is outside the scope of the thesis.

We have however made a distinction between the way that different patterns type is defined and instantiated. Patterns of type **process** and **presentation** have the attributes we have described and are defined and used identically. All patterns of these types are defined using SML functors. The reason is that we have identified these types of patterns as being parameterized modules of an abstract pattern. Therefore in NPDL patterns of these types are extensions of an abstract pattern and the use of the pattern in the design is a parameterized instantiation of the pattern's definition. The abstract basic pattern of this type has two parts that define some of its attributes. The first part is a set of the “screens” attribute (including their screen elements, authorizations and

connectors inside the module) and the “number of screens” attribute. Following is the signature of this part:

```
signature genericDesignSig =
sig
   val screens: scrset.set
   val numOfScreens: int
end
```

Signature of part 1

The second part is a set of the “module name” attribute (which module's design the pattern represents), the “set of requirements” attribute (i.e. which functional requirements are fulfilled by implementing this module) and the “reachable screens” attribute (i.e. which screens from outside modules can be reached from this module). Following is the signature of this part in NPDL:

```
signature propertiesDesignSig =
sig
   val moduleName: string
   val fulfillsRequirements : reqIDset.set
   val reachableScreens: scrIDset.set
end
```

Signature of part 2

As we will see later, each pattern of the types process and presentation implement these signature as well as add properties that are unique to them. They will all however be parameterized ML modules.

The third patterns type is **architectural patterns**. These patterns have a hierarchical type of connection between the modules that make up the pattern. For example, a university Web application is a pattern including modules such as: grades, courses information, course registration, personal calendar and so on. We chose to present the present this hierarchical structure as a tree structure in SML. The nodes of the tree are modules (the screens attribute of the pattern used for this module) and sub-trees are modules that are

embedded within a certain modules. For example, in a “course group” module, we would expect to find a course calendar of activities and to find a list of participants in the course. The syntax for constructing the pattern is as follows:

```
val <pattern name> = insertSon(<module screens>,<module screens>);
val <pattern name> = insertNode(<sub application>,<pattern name>);
```

```
datatype Application =
        Empty
      | Leaf of scrset.set
      | Node of scrset.set * Application list;

fun insertSon (x, Empty) = Leaf(x)
      | insertSon (x, Leaf(y)) = Node (y,[Leaf(x)])
      | insertSon (x, Node(y, app))   = Node(y,app@[Leaf(x)]);

fun   insertNode (x, Empty) = x
      | insertNode (x, Leaf(y)) = Node(y, [x])
      | insertNode (x, Node(y, app)) = Node(y,app@[x]);
```

Definition of an Architectural Pattern elements and functions in NPDL

A concrete example will be presented in the next chapter, regarding the CampusNet use case design.

Elements of Navigational Design Patterns in NPDL

In this section we demonstrate how to define the various elements of navigational design Patterns. These elements have been discussed previous chapters.

Screen Elements and Connectors

The screen elements are the visual elements that make up a web application screen, such as a link, button or input box. We have selected a subset of the most popular elements to define in NPDL, but of course any other element can be defined in the same manner

as demonstrated. The screen elements themselves are all structures with their unique attributes. Each element has an element-type (denoted by *val myType*) and a Name, which will be given by the pattern designer when instantiated in a screen. Each screen element also has unique attributes, e.g. a link has a target, which is the identifier of the screen in the application to which it leads, when clicked. This is an implementation of the connectors in NPDL within a module. A button screen-element has the attribute of the function it will activate when pressed. We present a portion of the NPDL library that implements the screen elements:

```
structure Link =
struct
   val myType = "link"
   type Name = string
   type Target = int  (*genericscreen*)
   type NewWin = bool
end

structure Button =
struct
   val myType = "button"
   type Name = string
   type Function = defaultFunction.Function
   type clickableOnEntry = bool
end

datatype screenElement =
          link of (Link.Name * Link.Target * Link.NewWin) |
           button of (Button.Name * Button.Function * Button.clickableOnEntry) |
           checkbox of (CheckBox.Name * CheckBox.Function * CheckBox.checkedOnEntry) |
           singleOption of (SelectSingleOption.Name * SelectSingleOption.DataSource) |
           multipleOptions of (SelectMultipleOptions.Name * SelectMultipleOptions.DataSource) |
           dataTable of (DataTable.Name * DataTable.DataSource) |
           displayText of (DisplayText.Name * DisplayText.DataSource) |
           inputbox of (InputBox.Name * InputBox.InputType * InputBox.isMasked);
```

Some Element Definitions and Screen Elements as defined in NPDL

Screen elements will be integrated into groups to form the individual components of a navigational design, i.e. Screens. Before that we introduce the NPDL implementation of authorizations, which are extensions to the patterns we have

introduced before.

Authorizations

Authorizations are implemented in NPDL to allow the designer to indicate which user groups can access which screens. The language provides a signature of the Authorization Group and a default implementation (*defaultAuthGroup*) where each Authorization Group is identified by a unique integer. Each group has the set of attributes as discussed in the previous chapter. Moreover, a set of default Authorization Groups is implemented, so that each screen can be associated easily with such as set.

```
signature ApplicationAuthorizationGroupSig =
sig
        type Id
        type Name
        type AllowedScreens
        type ForbiddenScreens
        type AuthGroup
        val GetAllowed: AuthGroup -> AllowedScreens
        val IsAllowed: (defaultScreen.Id * string) -> AuthGroup -> bool
end

structure defaultAuthGroup : ApplicationAuthorizationGroupSig =
struct
        type Id = int
        type Name = string
        type AllowedScreens = scrIDset.set
        type ForbiddenScreens = scrIDset.set
        type AuthGroup = (Id * Name * AllowedScreens * ForbiddenScreens)
        fun GetAllowed (id , n, asc, fsc) = asc
        fun IsAllowed sc ag = if scrIDset.member(sc, GetAllowed(ag)) then true else false
end

structure AuthGroupIDset = SetFct(type element = defaultAuthGroup.Id)
```

Authorization Signature and default Implementation and sets in NPDL library

Each actual authorization group (e.g. administrators), is instantiated by the navigational designer and associated with the set of screens it is allowed and not allowed to access. The codes 0 and 10000 are reserved for **no screens** (allowed or not allowed)

and **all screens** (allowed or not allowed). This issue has been discussed in the previous chapter and hence the implementation.

Screens

Screens are the basic building blocks of patterns. A screen in NPDL has an identifier, a name and elements that make-up the visualization of the screen. Note that when a screen is created (***newScreen*** method) it takes a module name (denoted as the *string* parameter). The reason for this is that the Id may appear several times in an application (for example, the main screen of a catalog pattern, which has the ID 300), but always in different modules. The Screen Identifier is set by the pattern designer, but the module parameter is set only when the pattern is instantiated by the Navigational designer. The latter setting is discussed in a following section about instantiation in more details.

Below is the NPDL signature and default implementation (***defaultScreen***) of screens. A set of default screens and screen identifiers has been provided as well.

```
signature ScreenSig =
sig
   type Id
   type Name
   type Elements
   type Screen
   val newScreen: Id -> Name -> Elements -> string -> Screen
   val ScreenName: Screen -> Name
end

structure defaultScreen : ScreenSig =
struct
   type Id = int
   type Name = string
   type Elements = (screenElement * real) list
   type Screen = (Id * Name * Elements * string)
   fun newScreen id n elem module = (id , n , elem , module)
   fun ScreenName  (id , n, elem, module) = n
```

```
end

structure scrset = SetFct(type element = defaultScreen.Screen)
structure scrIDset = SetFct(type element = (defaultScreen.Id * string))
```

Screen Signature and default Implementation as sets in NPDL library

Defining Basic Patterns

We now present the set of tools required to define the patterns themselves. In this section we demonstrate the way to define basic navigational patterns of type process and presentation, in NPDL. This part is relevant mainly for persons that define patterns, in order to publish in the patterns catalog. This catalog will then be accessed by designers who wish to use the patterns within their NPDL designs of Web applications.

```
functor loginFct(structure Usr: ApplicationUserSig structure GD: propertiesDesignSig): loginDesignSig =
struct
   structure Generic : genericDesignSig =
   struct
          val screens =       scrset.linsert([defaultScreen.newScreen 1 "login"
                                               [(inputbox("user ID",String,false),1.0),
                                                (inputbox("password",Numeric,false),2.0),
                                                (button("logon",(defaultFunction.newFunction "login" [("user
name","int"),("password","string")] [(1,"success"),(2,"fail")]),true),3.0),
                                                (checkbox("rememeber me",(defaultFunction.newFunction
"rememeberMe" [("user name","int")] [ ]),false),4.0)] GD.moduleName,
                                           defaultScreen.newScreen  2 "main"
                                                [(button("logout",(defaultFunction.newFunction "logout" []
                                                          [(1,"success"),(2,"fail")]),true),1.0)] GD.moduleNam
defaultScreen.newScreen 3 "logout" [(link("back to login",1,false),1.0)] GD.moduleName],
                              scrset.empty)
                      val numOfScreens = scrset.setsize(screens)
   end
   structure Properties = GD;
          structure User = Usr
   type login = (User.Id * User.Password)
   fun loginNewUser (u, p) = 1
   fun loginReturningUser p = 1
   fun failedLogin (u, p) = [(1,"incorrect password"),(2,"server not responding"),(3,"user inactive")]
   fun rememberMe u = User.getLogin(u)
end
```

Example of Defining the LOGIN Navigational Design Pattern (from NPDL Library)

Defining Complex Patterns using Combination

One of the important issues that NPDL addresses is the ability to define complex navigational patterns using the basic patterns. The construction method involves the combination of the basic patterns. For example, a **searchable catalog** pattern is a pattern that is, in fact, a combination of the independent patterns: **search** and **catalog**. A basic e-commerce pattern is a combination of a **searchable catalog** and **shopping basket** patterns.

Using **NPDL** the pattern designer can specify the screens of the patterns as inherited from more basic patterns, by specifying those patterns' names. This combination creates a link between the complex pattern and the inherited ones, so that the screen set is implicit, but can still be referred to using the same interface. The issue that the designer has to address is the internal navigation within the complex pattern. This is because the basic patterns handle the navigation exclusively within themselves. **NPDL** enables the designer to extend the screens with new components (such as links) or update existing components so that all the screens within the module are connected.

```
functor searchCatalogFct(structure Itm: ItemSig structure GD: propertiesDesignSig):
searchCatalogDesignSig =
struct
        structure CAT = catalogFct(structure Itm=Itm; structure GD=GD)
        structure SEAR = searchFct(structure Itm=Itm; structure GD=GD)
        structure Generic: genericDesignSig =
                struct
                        val screens = scrset.sinsert(SEAR.Generic.screens,CAT.Generic.screens)
                        val numOfScreens = scrset.setsize(screens)
  end
  structure Properties = GD
end
```

Defining a Searchable Catalog by combining Patterns - Example

In the example we see how a searchable catalog is defined by combining a basic catalog pattern (CAT) and a search pattern (SEAR). Note that the functor still adheres to the same structure as a basic pattern and can be further combined within even more complex pattern. Moreover, the basic patterns are simply referenced, so if they are updated, their properties and constructions are automatically propagated to this pattern, as well.

There are several important benefits to combining patters, rather than redefining existing patterns in more complex ones. These benefits (common to all software engineering best practices) are listed below:

- **Standardization:** when complex patterns inherit the structure of more basic ones, it is easier to learn and understand the more advanced patterns, by learning the basic building blocks.
- **Modularity**: the patterns definition process becomes more flexible and adaptable to changes and new 'best-practice' solutions
- **Reusability**: patterns that have been defined before and are useful in a more complex case are re-used speeding the definition process. This 'inheritance' is a contribution both to FPL and Navigational Design Patterns
- **Library Maintenance**: since complex patterns often are simply built from more basic patterns, any update to basic patterns automatically “escalates” through all the patterns that inherit its attributes.

Instantiation of Patterns

Once the patterns have been defined (in shared libraries or by the designer) they can be used by the designer of the Web application. This section is mainly aimed at navigational designers that use

(predefined) patterns within their NPDL design. NPDL provides the mechanism for a simple instantiation of the patterns to create entire modules in the application. The module navigational design then conforms to that of the pattern by this instantiation.

```
structure CourseItem = defaultItem;

structure CourseCatalogProperties : propertiesDesignSig =
struct
        val  moduleName = courseCatalogModuleName;
        val  fulfillsRequirements = (reqIDset.linsert([8],reqIDset.empty));
         val  reachableScreens = (scrIDset.linsert([(2,loginModuleName),
                                        (100,courseModuleName)],scrIDset.empty)
                                    );
end

structure courseCatalogModuleDesign = searchCatalogFct(structure Itm=CourseItem;structure
GD=CourseCatalogProperties);
```

Instantiating A Searchable Catalog Pattern as a Course Catalog Module

In the above example, the Course Catalog Module has been designed using the **Searchable catalog** complex pattern (*searchCatalogFct*). The navigational designer provides the properties: module name, which requirements are fulfilled (here requirement number 8) and the external screens that are reachable from this module (here screen 2 in the login module and screen 100 in the Course Module). This parameter (called GD) and the *CourseItm* Structure (representing a catalog of courses and defined as a default item[1]) are sent to the pattern functor to create the entire navigational design of the module. The pattern itself contains all the information about the screens, their structure and their internal connectivity scheme (through the use of links screen elements). The result produced by the SML interpreter mosml is shown:

---

1 *DefaultItem is an implementation of an abstract item that is common to several patterns in the NPDL Library, like catalog and basket*

```
structure CourseCatalogProperties :
{val fulfillsRequirements : set/29,
 val moduleName : string,
 val reachableScreens : set/28}
structure courseCatalogModuleDesign :
{structure CAT :
   {structure Generic : {val numOfScreens : int, val screens : set/27},
    structure Item :
      {type Id = int,
       type Value = int,
       type Description = string,
       type Item = int * int * string,
       val EQ : int -> int -> bool,
       val getItemId : int * int * string -> int,
       val newItem : int -> int -> string -> int * int * string},
    structure Properties :
      {val fulfillsRequirements : set/29,
       val moduleName : string,
       val reachableScreens : set/28},
    type catalog = set/38,
    val addItem : int -> set/38 -> set/38,
    val deleteItem : int -> set/38 -> set/38,
    val empty_catalog : set/38,
    val isMember : set/38 -> int -> bool,
    val isempty : set/38 -> bool,
    val numOfItems : set/38 -> int,
    val toList : set/38 -> int list},
 structure Generic : {val numOfScreens : int, val screens : set/27},
 structure Properties :
   {val fulfillsRequirements : set/29,
    val moduleName : string,
    val reachableScreens : set/28},
 structure SEAR :
   {structure Generic : {val numOfScreens : int, val screens : set/27},
    structure Item :
      {type Id = int,
       type Value = int,
       type Description = string,
       type Item = int * int * string,
       val EQ : int -> int -> bool,
       val getItemId : int * int * string -> int,
       val newItem : int -> int -> string -> int * int * string},
    structure Properties :
      {val fulfillsRequirements : set/29,
       val moduleName : string,
       val reachableScreens : set/28},
    type search = set/39,
    type param = string * string,
    val searchItem : string * (string * string) list * (int * string) list}}
```

Result of expanding the NPDL design by mosml

The expansion clearly shows how much information the design really has, although the NPDL syntax is very compact and clear. The developer can have access to this information, such as the screens of the sub-modules, the number of screens and variable types.

Diagnosis of Designs

Standardizing the design patterns and thus the designs themselves

presents the new possibility to perform diagnostics on the design. This is due to the reason that the format (or syntax) of any input (a design in NPDL) is known and therefore an algorithm to analyze required properties of the design can be written independently. Examples of diagnostics that may be performed are:

- screens that are reachable from many modules

This is important to check because it means that screens will be encountered often and their functionality should be efficient

- Is a screen reachable within a module?
- What is the shortest number of screens between 2 given screens?
- Are there screens that are not reachable for a certain user group?

More than defining all the possible diagnostics, **NPDL** focuses on providing a solid framework, so that the developers of such algorithms have all the required information encapsulated within the **NPDL** design.

```
(* screens that are reachable (= linked to) directly from multiple sources *)
fun   screenReachability sc mdname [ ] = 0
            | screenReachability sc mdname (Rscr::RScrList) =
if scrIDset.member((sc,mdname),Rscr) then 1 + screenReachability sc mdname RScrList
else screenReachability sc mdname RScrList

fun   multipleReachability sc mdname RScrList = (screenReachability sc mdname RScrList > 3);
```

Example of Diagnostic Code (From the NPDL Library)

The code example shows how to write code in order to check whether a screen is reachable from more than 3 modules. This indicates multiple entry points, which implies a need to make an efficient implementation of the screen. Using NPDL this property can be diagnosed after a design to improve instructions to the development team. Furthermore, the diagnostic code can be written independently from the design since the design is know to be in NPDL and the code can be reused.

Summary and Discussion

This chapter introduced NPDL, the language that can be used by navigational pattern designer, navigational designers of web applications and by persons who need to perform diagnostics of the design. We have shown how the properties that we required from the language in the previous chapter have been incorporated into the design of **NPDL**, so it has a broad basis to build upon and therefore increase the level of usefulness and thus of acceptance. The language in based on the research of Navigational Design Patterns and on the capabilities of SML, a functional programming language.

There are numerous benefits that we identified by designing NPDL in the way that we have. We list those advantages followed by a brief explanation:

- **Standardizing Designs** – the fact that we have chosen SML as a basis with a syntax and rules in place makes NPDL have these properties as well
- **Flexibility** – NPDL can still evolve within the existing framework. Also, SML has other properties that can further be incorporated into NPDL
- **Decouple Design Process** – each of the roles within the process has a part of NPDL that can be used separately, but in relation to the others. This means that the language supports the entire process, on one hand, but allows a parallel and independent work, on the other. This makes the process more efficient, which is very important when the application's design scales to complex levels.
- **Compactness –** the language is designed for all the parties involved in a way that the focus is on what needs to be conveyed. The syntax is designed so that how it is achieved can be described compactly and clearly, since it is intended for

complex application's design.

- **Type checking and validation** – the power of SML in type-checking and validating statements across the code (here, NPDL designs) is harnessed to validate the design as it progresses and provide type information to developers and other designers that are exposed to the NPDL final result.
- **Compatibility and Adaptability** – the use of a standard output from the *mosml* interpreter allows NPDL to fit well into other SML implementations. Furthermore, since the syntax is defined, an NPDL-based design can be adapted and converted into other formats, so it is an input to other systems. The formats can be, for example, XML or graph-based. Other systems can be HTML code-generators or visual tools.

The next chapter is a concrete example of designing a Web application using NPDL. The goal is to demonstrate the way in which the navigational designer is going to work when using NPDL – what is required and how it is done.

Chapter 5 - CampusNet Use Case

Campusnet, the DTU portal for staff and students, provides a test case for the thesis as a Web based application platform for examples and diagnostic purposes. In this chapter, we analyze the use case, list a set of navigational design patterns as they are used in CampusNet and propose an NPDL design of a portion of the application. This chapter demonstrates the thesis as it has been described in previous chapters and will be the basis for the practical part of the conclusions.

Use Case Analysis

The aim of the use case analysis is to explore the following issues:

1. What requirements have guided Campusnet designers?
2. What documentation exists for Campusnet design?
3. How do the design decisions relate to the requirements?
4. What problems do actual users face when using the Campusnet?
5. Do the problems occur due to lacking requirements or due to problems in the design?
6. How does the use of NPDL aid detecting or solving these problems?

Processes in the System

Some of the processes that have been identified and correspond to functional requirements are:

1. Secure login to the system
2. Entering a new event in the calendar
3. Ability to register and to delete registration to courses
4. Viewing and editing course participants list

Design Problems Identified

Some of the usability problems that have been identified were:
1. Inability to enter multiple meeting schedules
2. Slow login process
3. Slow course participant search
4. Entering messages forces a preview in a separate screen

CampusNet serves as a case study application. We describe the following processes in the system and the corresponding screens and functions. The first process is entering an activity to a calendar. The process is made up of activities (or steps) the user must follow in order to achieve the result. The navigational designer needs to map these activities into a set of screens, functions and navigational paths between the screens. An example of this mapping is shown at the table below:

P1: Enter Calendar Activity Process		
Activity	Screen	Function
Select language	LOGIN	selectLang(lang)
Enter user and password	LOGIN	GetUser(user, pwd)
Select group (links on left)	MAIN SCREEN	link
Select “calendar” (link)	GROUP SCREEN	link
Select “add new activity” (link)	CALENDAR SCREEN	AddActivity(groupID)
Enter activity details	ACTIVITY SCREEN	Manual data entry

P1: Enter Calendar Activity Process		
Click save	ACTIVITY SCREEN	SaveActivity(GroupID, details)
Click Logoff		Logout()

The second process is viewing an activity in the calendar. An example of this navigational designer's mapping is shown in the following table:

P2: View Calendar Activity Process		
Activity	Screen	Function
Select language	LOGIN	selectLang(lang)
Enter user and password	LOGIN	GetUser(user, pwd)
Select group (links on left)	MAIN SCREEN	link
Select date in monthly calendar	CALENDAR SCREEN	getDisplayDate(date)
Click on activity in weekly display	CALENDAR SCREEN	displayActivity(ID)
view activity details	ACTIVITY SCREEN	---
Click close	ACTIVITY SCREEN	CloseWindow()
Click Logoff		Logout()

Another process is sending a message to participants in a course. An example of the activities mapping is shown below:

P3: Message to Group Participants Process		
Activity	Screen	Function
Select language	LOGIN	selectLang(lang)
Enter user and password	LOGIN	GetUser(user, pwd)
Select group (links on left)	MAIN SCREEN	openGroupScreen(groupID)

Click messages	GROUP SCREEN	Link
Click compose message	MESSAGES SCREEN	Link
Enter message data	MESSAGE SCREEN	Manual
Press “see message”	MESSAGE SCREEN	PreviewMessage(data)
Click “send”	MESSAGE PREVIEW SCREEN	SendMessage(contacts, data)
Click Logoff		Logout()

User Groups in CampusNet:

- Superuser
- Administrator
- Author
- User

This identification of requirements, the mapping to actual screens, functions and navigations is the primary task of the navigational designer of CampusNet. This is the link between the functional requirements of the CampusNet fuure users and the solution provided by the navigational designer to the way they will interact with the application. This abstract mapping will now be presented using NPDL, so it is clear how the proposed solution is applied to a real-life Web application.

Functional Requirements in CampusNet

The first step CampusNet the navigational designer follows is defining the functional requirements of the application using NPDL. We present some of the requirements for CampusNet, some of which are used further in the example.

```
(* Campusnet Functional Requirements Definition *)
val fr = FunctionalRequirements.insert((1,"register to courses"),
                FunctionalRequirements.insert((2,"view student grades"),
                FunctionalRequirements.insert((3,"view course participants list"),
                FunctionalRequirements.insert((5,"secure login to the system"),
                FunctionalRequirements.insert((6,"allow viewing course-participant details"),
                FunctionalRequirements.insert((4,"send messages to course participants"),
                FunctionalRequirements.insert((7,"track course related activities in course and personal
calendar"),
                FunctionalRequirements.insert((8,"the system must enable adding, viewing and
searching for courses"),
                FunctionalRequirements.insert((10,"allows users to access main modules, such as
messages and email from all screens"),FunctionalRequirements.empty)))))))));
```

Defining Functional Requirements of CampusNet in NPDL

Navigational Design Patterns in CampusNet

After showing how the functional requirements are defined in NPDL, we show the navigational design patterns we have identified in CampusNet. In practice, it is the navigational designer's task to relate between the problems that the functional requirement demand solving and the corresponding navigational design patterns that offer a best-practice solution. The format for the patterns presentation is similar. We present each of the screens, including the data and operations (screen elements), screenshots of the actual module in CampusNet and an NPDL design of the module, which is an instantiation of the presented pattern.

- **Login pattern**

The login pattern handles the problem of a secure login to a system using usually a user name and password. The instantiation in this case is to the login module in CampusNet.

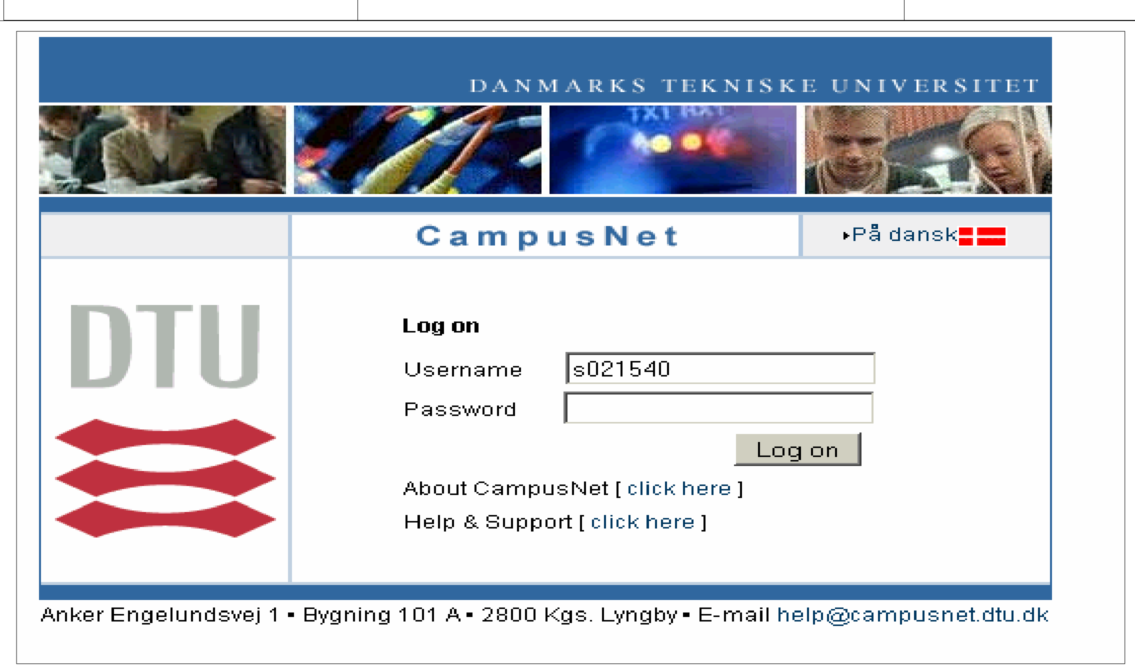
screenID: 1000
Name: "login screen (English)"

Data:

Name	Data type	Navigation
User name	string	%
Password	String	%
Language	DANSK	⊥

Operations:

Name	Parameters	Navigation
Log on	User name, password	n → 1100
Help		n → 2001
About		n→ 2002



```
val  loginModuleName = "campusnet login";

(* campusnet login *)
(* campusnet user - example of overriding a defaultUser structure if required *)
structure CampusnetUser : ApplicationUserSig =
struct
                        type Id = int
       type Password = string
       type authorizationGroups = AuthGroupIDset.set
       type user = (Id * Password * authorizationGroups)
       fun newUser id = (id, "****",AuthGroupIDset.empty)
       fun hasAuthorization (id,p,ag) a = if AuthGroupIDset.member(a,ag) then true else false
                fun addAuthorization (id,p,ag) a = (id,p,AuthGroupIDset.insert(a,ag))
                fun deleteAuthorization (id,p,ag) a = (id, p, AuthGroupIDset.delete(a,ag))
       fun printUser (id,p,ag) = id::AuthGroupIDset.toList(ag)
       fun getLogin (id,p,ag) = (id, p)
end

structure loginProperties : propertiesDesignSig =
struct
                        val  moduleName = loginModuleName;
                        val  fulfillsRequirements = (reqIDset.linsert([5],reqIDset.empty));
                        val  reachableScreens = (scrIDset.insert((200,calendarModuleName),
 scrIDset.insert((200,myCalendarModuleName),
 scrIDset.insert((100,courseModuleName),scrIDset.empty))));
end
structure loginModuleDesign = loginFct(structure Usr=CampusnetUser; structure GD=loginProperties);
```

Login Module Design in NPDL

- **Calendar pattern**

The Calendar pattern handles the problem of entering and managing activities of the user on a schedule basis. The instantiation in this case is to the personal Calendar and course Calendar modules in CampusNet.

screenID: 1001
Name: "calendar overview"
Data:

Name	Data type	Navigation
selector		
CalendarDisplay	SelectedDate: Date	

Operations:

Name	Parameters	Navigation
InsertNewActivity	Current date	n → 1002
ViewActivityDetails	ActivityID: numeric -> Activity	n → 1005
GoTo	month*year	↺
AppointmentsInThisWeek		inactive
UpcomingAppointments	date	→ 1003
AllAppointments		→ 1004

Calendar: Ziv Yosef Shapira
Show: Appointments in this week | Upcoming appointments | All appointments

Monday 09/02/2004 - Sunday 15/02/2004
Add new activity
Feb 2004 Go to

February 2004

Mon	Tu	We	Th	Fr	Sa	Sun
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
1	2	3	4	5	6	7

	Monday (09-02)	Tuesday (10-02)	Wednesday (11-02)	Thursday (12-02)	Friday (13-02)	Saturday (14-02)	Sunday (15-02)
All day							
07:00							
:15							
:30							
:45							
08:00				08:00 - 12:00 02417 Tidsrækkeanalyse Forelæsning/grupperegning Bygn. 321, Rum 033/Bygn.303, Rum 43			
:15							
:30							
:45							
09:00							
:15							
:30							
:45							
10:00							
:15							
:30							
:45							
11:00							
:15							
:30							
:45							
12:00							
:15							

screenID: 1002
Name: "Insert new activity"
Data:

Name	Data type	Navigation
startDate	{1-31}/{1-12}/{2000-2006}	╧
StartTime	{0-23}/{00,05,10,15,…,55}	╧
Duration	{0-23}/{00,05,10,15,…,55}	╧
Place		%
Participants		%
Subject		%
Description		%
All day		≈

Operations:

Name	Parameters	Navigation

Save	StartDate:Date StartTime: time Duration: numeric Place: String Participants: String Subject: String Description: String -> Activity	■

CampusNet - Netscape

Insert new activity

Date 16 2 2004

Starttime: 12 :00

Duration: 0 :00 All day

Place:

Participants:

Subject:

Body:

Save

screenID: 1003
Name: “upcoming appointments”
Data:

Name	Data type	Navigation
Date and time	StartDate:Date StartTime: time Endtime: time (calculated)	
details	Place: String Participants: String Subject: String Description: String	

Operations:

Name	Parameters	Navigation
InsertNewActivity		n → 1002
AppointmentsInThisWeek		→ 1001
UpcomingAppointments	date	inactive
AllAppointments		→ 1004

Thursday **26/02-2004** 08:00-12:00	**Forelæsning/grupperegning** Place: Bygn. 321, Rum 033/Bygn.303, Rum 43 Participants: 02417 Tidsrækkeanalyse - Dynamiske systemer - Karakteristika for Dynamiske Systemer - Lineære dynamiske systemer - Frekvens- og tidsdomænebeskrivelser - Overførings- og responsfunktioner - z-transformationen - Laplace-transformationen Ref. A Exercise: X8, 29
13:00-14:00	**weekly thesis status** Place: MRH OFFICE Participants: ZSH Thesis
Thursday **04/03-2004** 08:00-12:00	**Forelæsning/grupperegning** Place: Bygn. 321, Rum 033/Bygn.303, Rum 43 Participants: 02417 Tidsrækkeanalyse - Forskydningsoperatorer - Stokastiske processer - Momentfunktioner Ref. A. Exercise: EDB opogave nr. 1
13:00-14:00	**weekly thesis status** Place: MRH office Participants: ZSH Thesis weekly thesis status meeting
Thursday	

ScreenID: 1004
Name: “all appointments”
Data:

Name	Data type	Navigation
Date and time	StartDate:Date StartTime: time Endtime: time (calculated)	
details	Place: String Participants: String Subject: String Description: String	

Operations:

Name	Parameters	Navigation
InsertNewActivity		n→ 1002
AppointmentsInThisWeek	date	→ 1001
UpcomingAppointments	date	→ 1003
AllAppointments		inactive

Thursday **26/02-2004** 08:00-12:00	**Forelæsning/grupperegning** Place: Bygn. 321, Rum 033/Bygn.303, Rum 43 Participants: 02417 Tidsrækkeanalyse - Dynamiske systemer - Karakteristika for Dynamiske Systemer - Lineære dynamiske systemer - Frekvens- og tidsdomænebeskrivelser - Overførings- og responsfunktioner - z-transformationen - Laplace-transformationen Ref. A Exercise: X8, 29
13:00-14:00	**weekly thesis status** Place: MRH OFFICE Participants: ZSH Thesis
Thursday **04/03-2004** 08:00-12:00	**Forelæsning/grupperegning** Place: Bygn. 321, Rum 033/Bygn.303, Rum 43 Participants: 02417 Tidsrækkeanalyse - Forskydningsoperatorer - Stokastiske processer - Momentfunktioner Ref. A. Exercise: EDB opogave nr. 1
13:00-14:00	**weekly thesis status** Place: MRH office Participants: ZSH Thesis weekly thesis status meeting
Thursday	**Forelæsning/grupperegning**

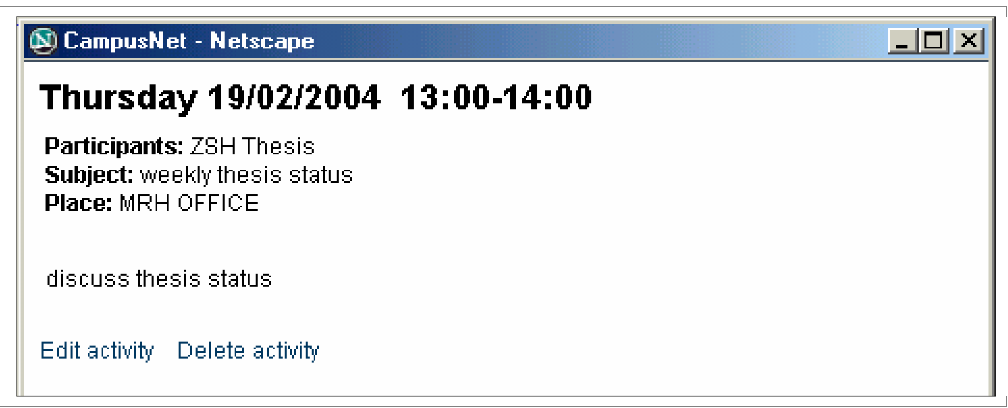
ScreenID: 1005
Name: "view activity"
Data:

Name	Data type	Navigation
Date and time	StartDate:Date StartTime: time Endtime: time (calculated)	

details	Place: String Participants: String Subject: String Description: String	

Operations:

Name	Parameters	Navigation
EditActivity	ActivityID: numeric	n→ 1002
Delete activity	ActivityID: numeric	→ 1001




```
val  myCalendarModuleName = "MY Calendar";
val  calendarModuleName = "course activities calendar";

(* campusnet  course activities in calendar *)
structure CampusnetActivity = defaultActivity;
structure ActivityCalendarProperties : propertiesDesignSig =
struct
            val  moduleName = calendarModuleName;
            val  fulfillsRequirements = (reqIDset.linsert([7],reqIDset.empty));
    val  reachableScreens = (scrIDset.empty);
end

structure calendarModuleDesign = calendarFct(structure Act=CampusnetActivity; structure
GD=ActivityCalendarProperties);
```

```
(* campusnet  personal activities in calendar *)
structure PersonalActivityCalendarProperties : propertiesDesignSig =
struct
    val  moduleName = myCalendarModuleName;
    val  fulfillsRequirements = (reqIDset.linsert([7],reqIDset.empty));
    val  reachableScreens = (scrIDset.insert((2,loginModuleName),
```

```
scrIDset.insert((100,courseModuleName),scrIDset.empty)));
end

structure myCalendarModuleDesign = calendarFct(structure Act=CampusnetActivity; structure
GD=PersonalActivityCalendarProperties);
```

Course Calendar and Personal Calendar Modules Design in NPDL

- **Basket Pattern**

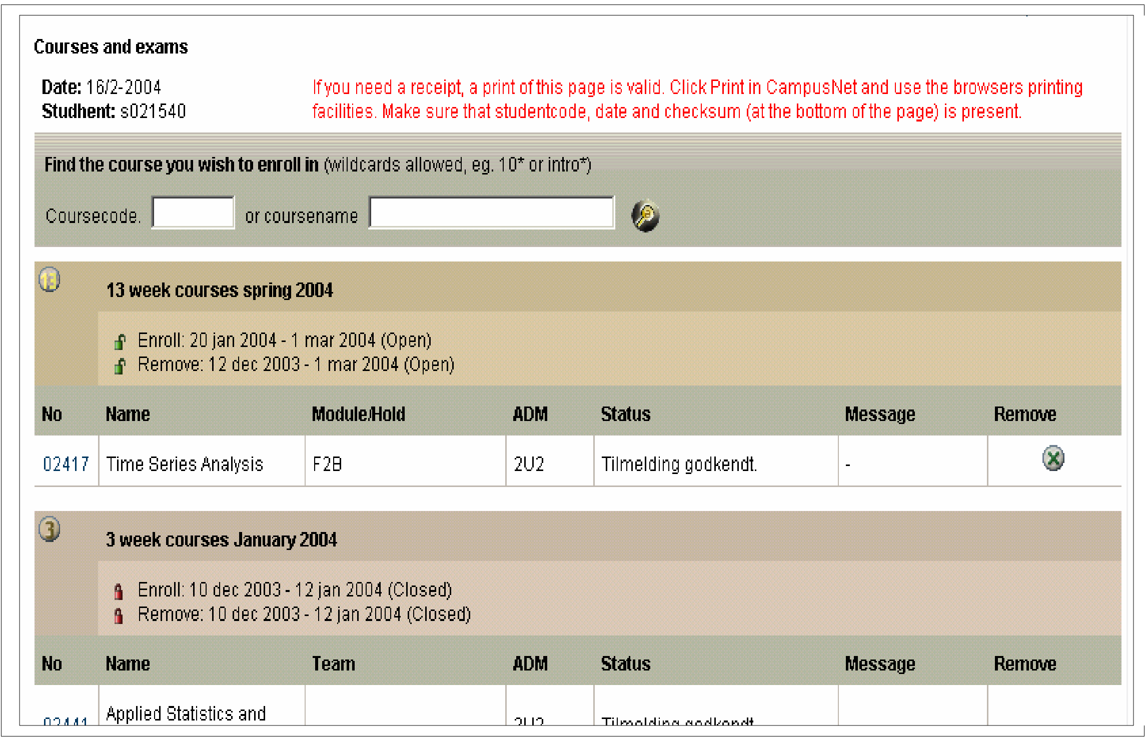
The basket pattern handles the problem of collecting items of a certain type and performing some operation on the entire set of items. The instantiation in this case is to the course registration module in CampusNet. In this case the pattern was not used in the actual system, but is used in the design we are making in order to present a possible improvement that solves one of the identified problems. In this pattern the items are courses and the operation is registration.

screenID: 1011
Name: "registration overview"
Data:

Name	Data type	Navigation
period		
Course	code: numeric name: string module: list admin: list status: string message: string	

Operations:

Name	Parameters	Navigation
Remove	PeriodID: numeric CourseID: numeric	n→ 1012
FindCoursesByID	*	n→ 1013
FindCoursesByName	*	→ 1014
Register	CourseID or CourseName	→ 1012



screenID: 1012
Name: "acknowledgment"
Data:

Name	Data type	Navigation
Message	String	▓

Operations:

Name	Parameters	Navigation
Back to list		→ 1011

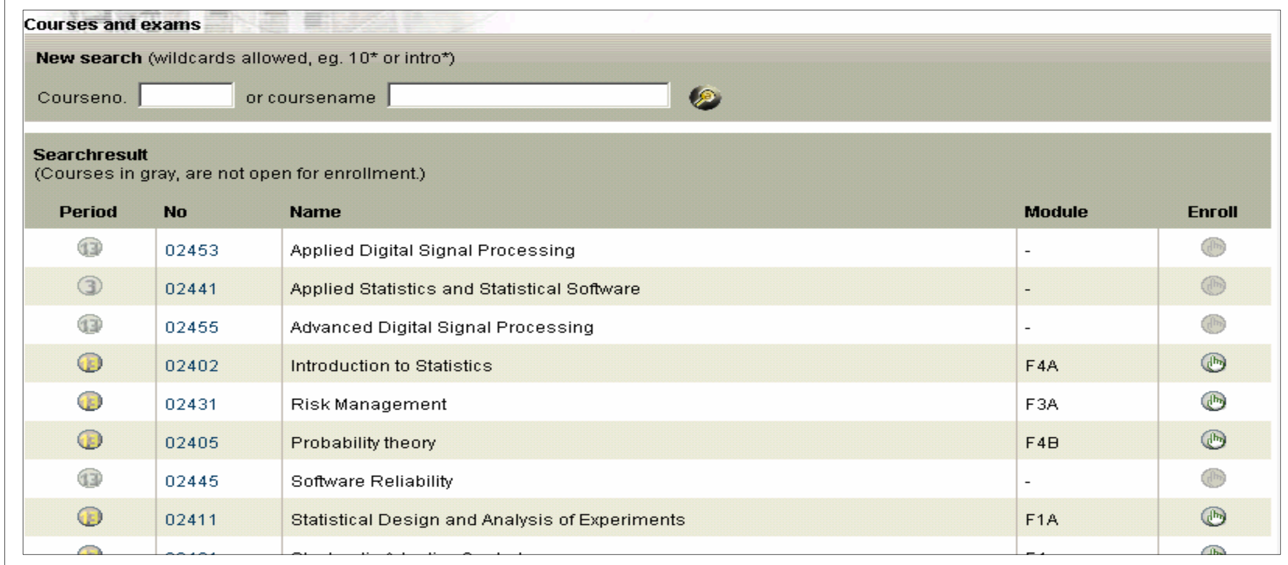
screenID: 1013
Name: "Find Course By ID"
Data:

Name	Data type	Navigation
period		

course	code: numeric name: string module: list	

Operations:

Name	Parameters	Navigation
Enroll	PeriodID, CourseID	n→ 1012
FindCoursesByID	CourseID: string *	
FindCoursesByName	CourseName: string *	→ 1014



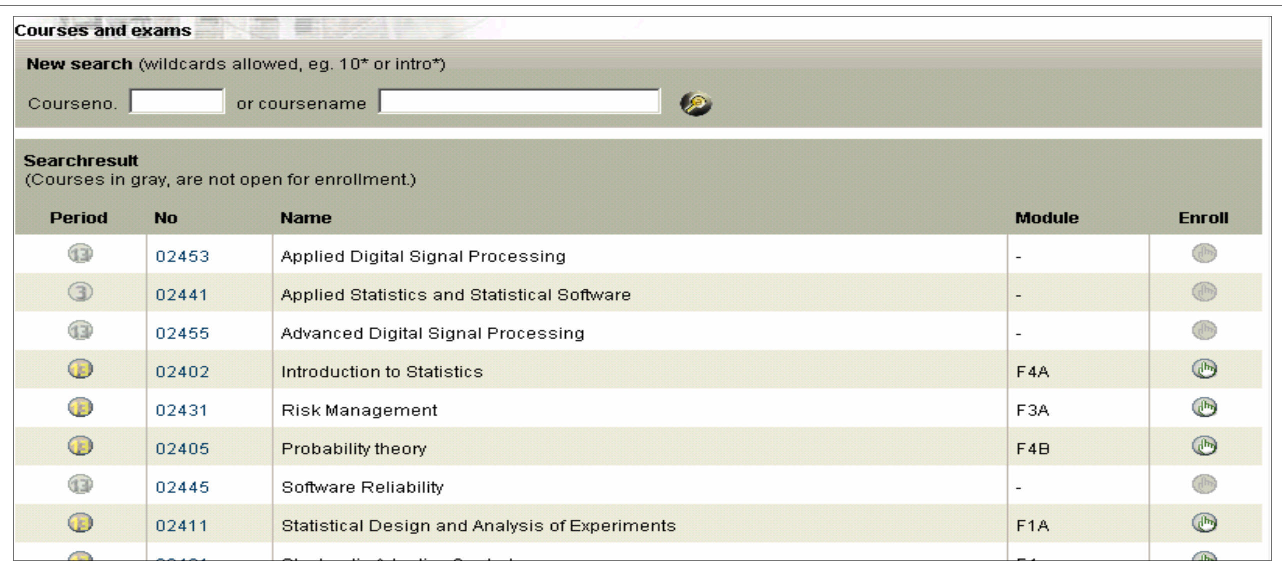
screenID: 1014
Name: “Find Course By Name”
Data:

Name	Data type	Navigation
period		
course	code: numeric name: string module: list	

Operations:

Name	Parameters	Navigation
Enroll	PeriodID, CourseID	n→ 1012

FindCoursesByID	CourseID: string *	→ 1013
FindCoursesByName	CourseName: string *	



Courses and exams

New search (wildcards allowed, eg. 10* or intro*)

Courseno. or coursename

Searchresult
(Courses in gray, are not open for enrollment.)

Period	No	Name	Module	Enroll
	02453	Applied Digital Signal Processing	-	
	02441	Applied Statistics and Statistical Software	-	
	02455	Advanced Digital Signal Processing	-	
	02402	Introduction to Statistics	F4A	
	02431	Risk Management	F3A	
	02405	Probability theory	F4B	
	02445	Software Reliability	-	
	02411	Statistical Design and Analysis of Experiments	F1A	

```
val  courseModuleName = "course registration module";

(* campusnet course Registration Module *)
structure CourseItem = defaultItem;
structure CourseProperties : propertiesDesignSig =
struct
        val  moduleName = courseModuleName;
        val  fulfillsRequirements = (reqIDset.linsert([1],reqIDset.empty));
        val  reachableScreens = (scrIDset.insert((2,loginModuleName),
                        scrIDset.insert((300,courseCatalogModuleName),
                        scrIDset.insert((200,calendarModuleName),scrIDset.empty)
                                )));
end

structure courseModuleDesign = basketFct(structure Itm=CourseItem; structure GD=CourseProperties);
val courseModulefl = courseModuleDesign.FuncList [
                        defaultFunction.newFunction "register" [("int","semester season"),("int","semester
year"),("int","student id")] [(1,"approved"),(2,"full"),(3,"not given this semester")],
                        defaultFunction.newFunction "print" [("int","printer ID"),("int","number of copies")]
[(1,"course info printed"),(2,"course info unavailable"),(3,"printer not responding")],
                        defaultFunction.newFunction "unregister" [("int","student ID")] [(1,"unregister
successful"),(2,"unregister failed")]];
```

Course Registration Design in NPDL as Basket Pattern

- **Personal Address Book pattern**

The address book pattern handles the problem of managing personal contacts and being able to search them effectively when needed. The instantiation in this case is to the Address Book module in CampusNet.

screenID: 1021
Name: “addresses overview”
Data:

Name	Data type	Navigation
Details	Name Phone Mobile Email Address website	%
Comment	String	%

Operations:

Name	Parameters	Navigation
Edit	ContatctID: numeric	n → 1022
InsertNewContanct		■ (1022)
SortBy	Given Name Last name Phone Mobile Email Address	⤸

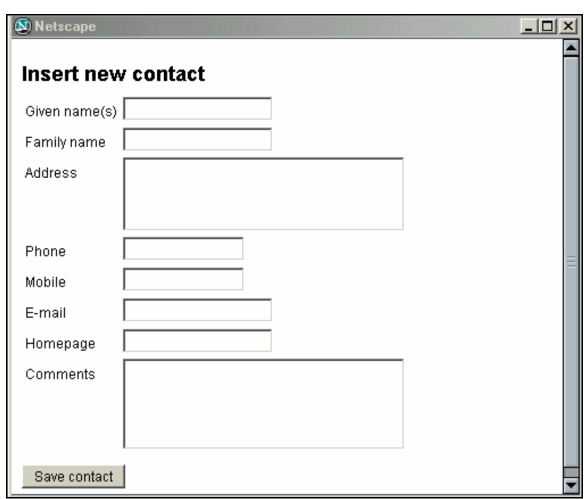
screenID: 1022
Name: “contact details”
Data:

Name	Data type	Navigation
Given name	String	%
Family name	String	%
Address	String	%

Phone	numeric	%
Mobile	numeric	%
Email	String	%
Homepage	URL	%
Comment	String	%

Operations:

Name	Parameters	Navigation
Save	contactDetails	■ (1021)



This section has demonstrated the mapping of the navigational designer, the similar process that would follow if using NPDL and the result in the actual application after implementation. The goal is to emphasize the compact and clear way that NPDL captures both the requirements and the solution the designer wishes to convey to the developers.

Other Modules

Some modules of CampusNet were designed using NPDL in order to demonstrate a practical example of using the language. Following the NPDL code that is the design without a graphical

representation of these modules. The modules are: Course participants, Course Catalog and Groups.

```
val  courseCatalogModuleName = "course catalog";
val  groupModuleName = "CN Groups (courses, DTU, user)";
val  courseParticipantModuleName = "course participants";

(* campusnet  course participants *)
structure CourseParticipant = defaultItem;
structure CourseParticipantProperties : propertiesDesignSig =
struct
      val  moduleName = courseParticipantModuleName;
      val  fulfillsRequirements = (reqIDset.linsert([3,6],reqIDset.empty));
      val  reachableScreens = (scrIDset.empty);
end

structure courseParticipantModuleDesign = catalogFct(structure Itm=CourseParticipant;structure
GD=CourseParticipantProperties);

(* campusnet  course catalog *)
structure CourseCatalogProperties : propertiesDesignSig =
struct
      val  moduleName = courseCatalogModuleName;
      val  fulfillsRequirements = (reqIDset.linsert([8],reqIDset.empty));
      val  reachableScreens =
(scrIDset.linsert([(2,loginModuleName),(100,courseModuleName)],scrIDset.empty)
                                                             );
end
structure courseCatalogModuleDesign = searchCatalogFct(structure Itm=CourseItem;structure
GD=CourseCatalogProperties);

(* Campusnet Group Module Design *)
structure groupModuleDesign =
struct
   structure Generic: genericDesignSig =
   struct
            val screens = scrset.linsert([defaultScreen.newScreen 10000 "Group Welcome"
[(displayText("welcome...",""),1.0),(link("search again",400,false),2.0)] groupModuleName],scrset.empty);
            val numOfScreens = scrset.setsize(screens);
   end
   structure Properties =
   struct
     val  moduleName = groupModuleName;
     val  fulfillsRequirements = reqIDset.empty;
     val  reachableScreens =
(scrIDset.linsert([(2,loginModuleName),(100,courseModuleName)],scrIDset.empty));
          end
end
```

NPDL Design of modules: participants, Course Catalog and Groups

The entire application portion we have designed was implemented as an architectural design pattern, which is a **Univesity Protal** pattern. The code and the result of printing this design are presented below:

```
(* designing the entire application as an architectural design pattern *)
val campusnetDesign = insertSon(myCalendarModuleDesign.Generic.screens,
Leaf(loginModuleDesign.Generic.screens));
val campusnetDesign =
insertNode(Node(courseModuleDesign.Generic.screens,[Leaf(courseCatalogModuleDesign.Generic.screen
s)]),campusnetDesign);
val campusnetDesign = insertNode(Node(groupModuleDesign.Generic.screens,
[Leaf(calendarModuleDesign.Generic.screens),Leaf(courseParticipantModuleDesign.Generic.screens),Leaf
(courseParticipantModuleDesign.Generic.screens)]), campusnetDesign);
```

CampusNet architectural Design in NPDL

```
########
(1,campusnet login,login)
(2,campusnet login,main)
(3,campusnet login,logout)

    (200,MY Calendar,Monthly display)
    (203,MY Calendar,weekly display)
    (204,MY Calendar,daily display)
    (201,MY Calendar,Display Activity Details)
    (202,MY Calendar,Enter New Activity)

    (100,course registration module,items list)
    (101,course registration module,function parameters)
    (102,course registration module,Feedback)

        (400,course catalog,search parameters)
        (401,course catalog,search results)
        (300,course catalog,items list)
        (301,course catalog,view item details)
        (302,course catalog,add new item)

    (10000,CN Groups (courses, DTU, user),Group Welcome)

        (200,course activities calendar,Monthly display)
        (203,course activities calendar,weekly display)
        (204,course activities calendar,daily display)
        (201,course activities calendar,Display Activity Details)
        (202,course activities calendar,Enter New Activity)

        (300,course participants,items list)
        (301,course participants,view item details)
        (302,course participants,add new item)

        (300,course participants,items list)
        (301,course participants,view item details)
        (302,course participants,add new item)

##########[closing file "campusnetDesign.sml"]
```

CampusNet design print-out of the NPDL code

The printing reveals the hierarchy of the modules, shown with the screens identifiers, module name and screen names, within the

CampusNet application. This is a visualizations of the levels in the application and is clearly visible to the designer when using NPDL.

Authorizations

Authorizations have been handled in NPDL. We have identified 3 groups: administrators, students and lecturers. Each group has been assigned the screens it may and may not view. For example, **Administrators** can view all screens, so they have been assigned the code 10000 (reserved for this purpose) in the “allowed Screens” group (*adminasc*) and students were denied the screen identified as 200 in the course calendar module (their forbidden screens group is called: *studfsc*). The 3 groups were then created using the newAuthGroup command. The full code is presented below:

```
(********************* AUTHORIZATION DEFINITIONS **************************)
val adminasc = scrIDset.insert((10000,"all"), scrIDset.empty);
val adminfsc = scrIDset.empty;
defaultAuthGroup.newAuthGroup 1 "admin" adminasc adminfsc;

val studasc = scrIDset.linsert([(1,loginModuleName),(2,loginModuleName),(3,loginModuleName),
        (100,courseModuleName),(101,courseModuleName),(102,courseModuleName)], scrIDset.empty);
val studfsc = scrIDset.insert((200,calendarModuleName), scrIDset.empty);
defaultAuthGroup.newAuthGroup 2 "students" studasc studfsc;

val lecasc =
scrIDset.linsert([(1,loginModuleName),(200,calendarModuleName),(200,myCalendarModuleName)],
scrIDset.empty);
val lecfsc = scrIDset.empty;
defaultAuthGroup.newAuthGroup 3 "lecturers" lecasc lecfsc;
```

Authorization Definition in CampusNet Using NPDL

Summary and Discussion

This chapter introduced an example of designing a Web University

Portal using NPDL. The main idea was to present the applicability of the NPDL design concepts. An architectural pattern, such as this one, can actually be defined by several pattern architects in different ways, so it is not confined, as other patterns are not, to a single definition. The definitions vary in the screen structure, number of screens or the navigational design between the screens. Each of the patterns has of course many instantiations within specific designs that use them. For example, CampusNet design may be an instantiation of an entire University Portal Pattern or made from instantiations of smaller patterns for the different modules.

We have chosen to present both the design of existing patterns that we have identified in CampusNet as they are, for example calendars and course participants. However, we also present designs of some modules based on other patterns we found more fitting. For example, the registration process is not based on the basket pattern in the actual system, but our NPDL design shows how this can be used. We believe this solution provides a more robust and standard design, since it is based on known patterns.

The problems we have identified in the existing design are described in the introduction to this thesis. As a reminder they were: that after logging in the user could not change the chosen interface language. The other one was that the course search and selection are in a completely separated application that the registration process. Moreover the registration can be done to each selected course individually and not collectively. By using NPDL and patterns, we have can discover the language identification problem by explaining the design diagnosis process. For solving the registration design problem we followed these steps:

- Defined a **Catalog with Search** Pattern in NPDL

- Used the **Catalog with Search** and **Basket** patterns in CampusNet's NPDL design of the Course Registration module

We achieved an improved navigational design for the module. The following user interactions are now possible:

- Presents the courses to the user in a searchable catalog in the application
- Allows a selection of one or more courses into a dedicated list (basket)
- Enables registration within the system to one or more of the courses in the list at the same time, by selecting the register operation
- The user (student) receives feedback on the registration result of each course. The feedback is similar to the one given in the actual CampusNet, so there is no loss of information

We see this as a viable and robust solution to the problems identified and attribute the solutions of both problems to the use of NPDL in our use case design.

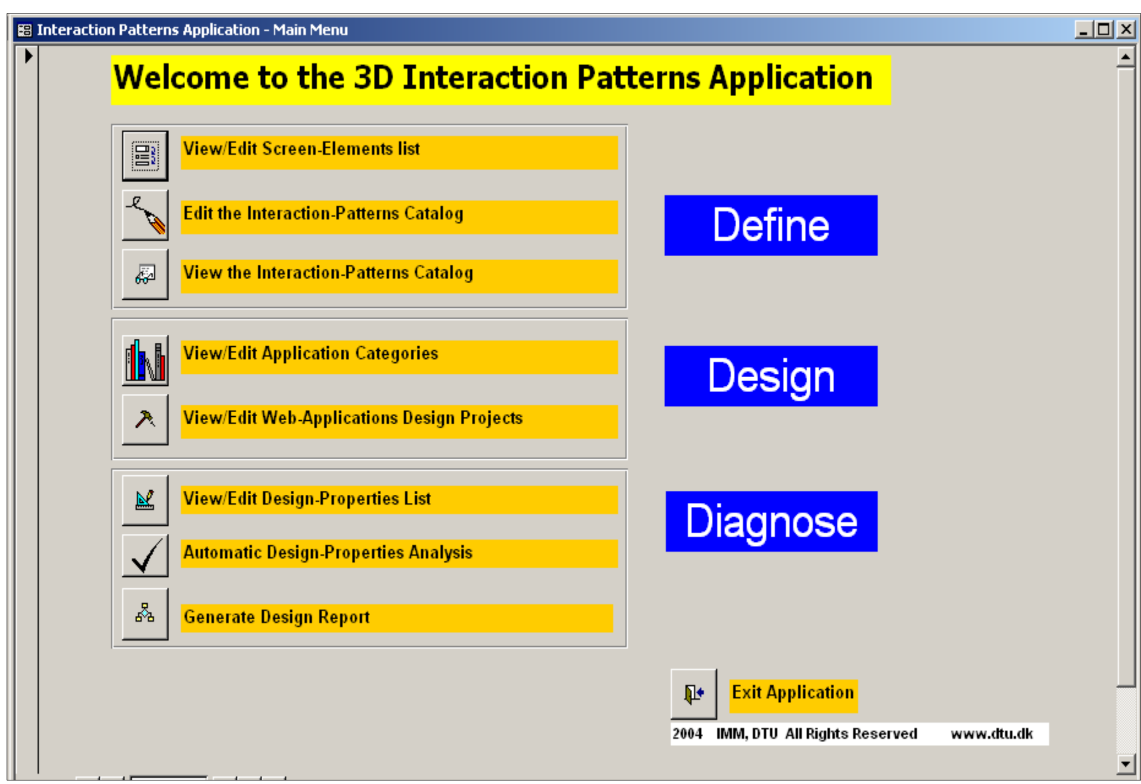
Chapter 6 - Visual Tool for Designers

This chapter presents NPDL, the language described in the previous chapter, as it may be used with a visual tool for navigational Designers. This approach presents an alternative to the use of NPDL with command lines, as presented, but still retaining the language's design principles. A visualization of the language's functionality can make it more accessible to designers without programming skills. The ideas presented in this chapter will add to the discussion and conclusions that will be introduced in the next chapter.

The application follows the navigational design process: defining patterns, designing Web applications using patterns and diagnosing the design. All three parts use NPDL as the underlying basis. The application is hence called "**The 3D Application**" (Define, Design and diagnose). We have designed the application based on client-server architecture, using Microsoft™ Access 2000.

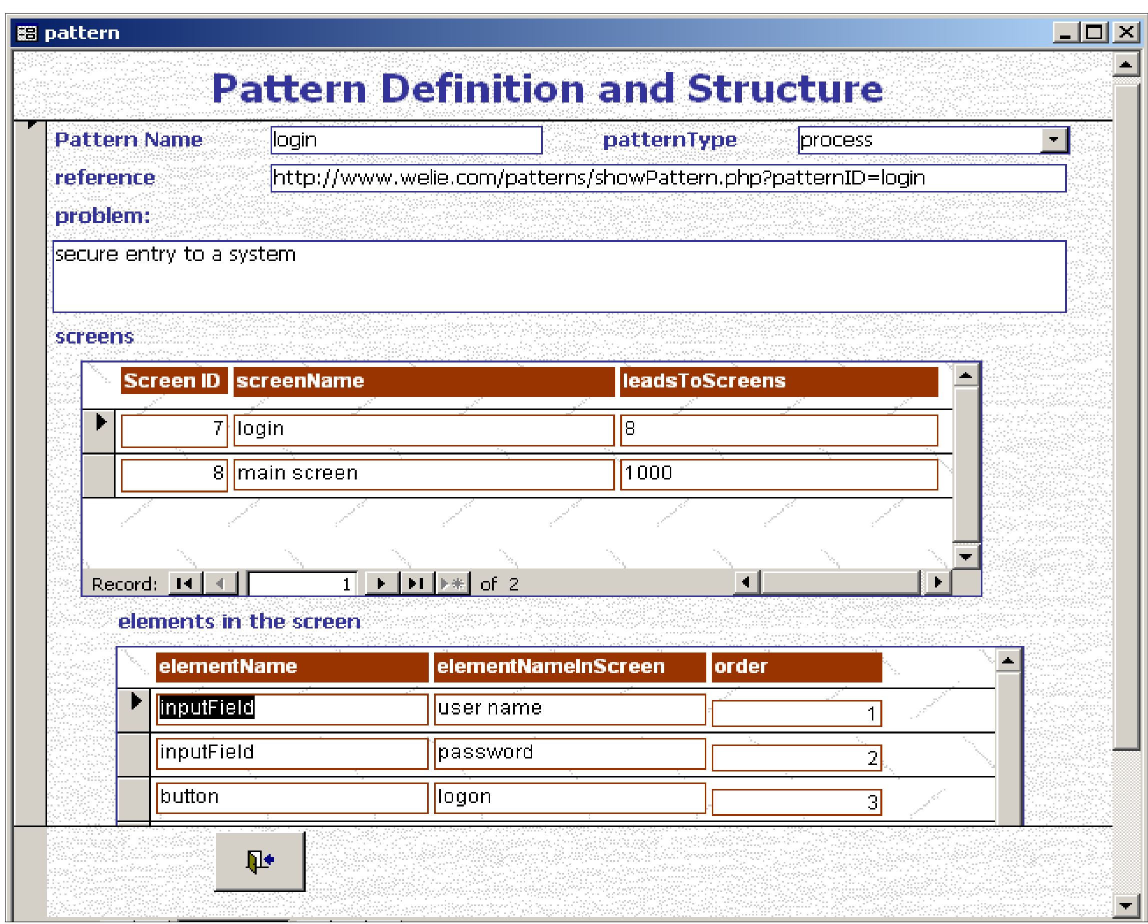
Main Functionalities

The main functionalities behind the 3D application are introduced in the main screen (shown below).



Main Screen of the 3D application

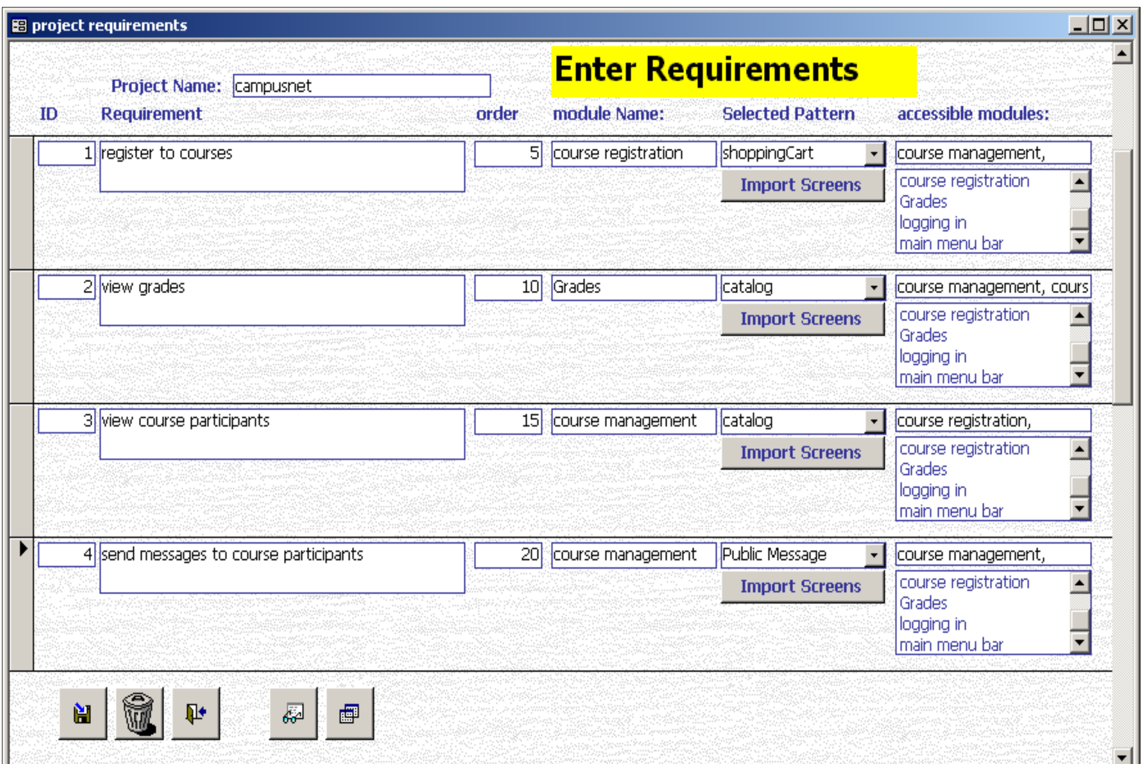
The first part of the application handles the functionalities required by the **pattern designers**. The application enables them to view and edit the screen elements (those defined in NPDL) and define navigational design patterns using these elements within screens. Referring to our description of the LOGIN pattern, the pattern's attributes, screens and screen-elements within each screen are defined. The corresponding result of defining this pattern in the 3D application will be seen as follows:



Login Pattern as defined in the 3D application

The second part of the application handles the functionalities required by the **Web Application Navigational designers**. They are able to view categories of applications, such as Customer Relationship Management (CRM) or University Portals. Mainly, however, they can create Navigational Designs using the patterns that have been defined, either by them or by others. The design follows the same guidelines dictated by NPDL design, i.e. Entering the Functional Requirements of the application, relating them to modules (given names) and patterns that represent the design of the modules (names of pattern), as chosen by the designer. The 3D application provides an easy access to the patterns catalog, so the

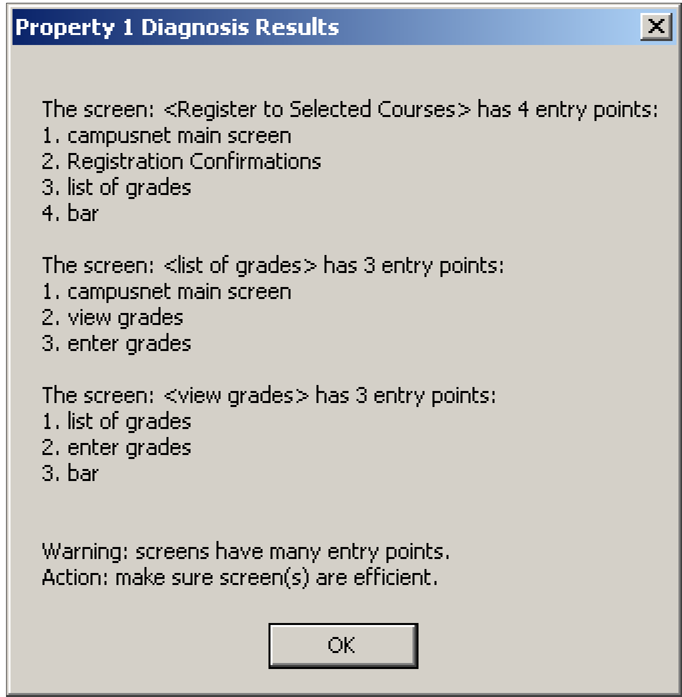
selection process is simple and based on the information provided by the patterns designers. The main idea and contribution is that the navigational designer can relate between the concrete requirements to a pattern that solves a problem, which is an abstraction of this requirement. For example, in CampusNet as concrete requirement to enables students to select courses (items) and register to them as a group (operation on items), can be related to the **basket pattern**. When a relation has been made, the designer can use the "Import Screens" button to instantiate the pattern within the specific design. This pattern provides a solution to the abstract problem provided here in parenthesis for clarity. The screen that encapsulates these functions is shown below:



Functional Requirements and Patterns for CampusNet (Partial View)

The designer can then go on to name the default screen elements, add or delete some of them to fit the design. The application

maintains the flexibility of the language, i.e. the degree in which the navigational designer adheres to the selected pattern. The third part is dedicated to the diagnosis of design, mainly used by **quality assurance** (QA) engineers. The contribution is in the ability to identify problems during the design phase, rather than after implementation, thus leading to more robust systems and efficient development process. The modules enable to define properties which are going to be investigated and view some that have been defined already. The QA engineer can execute an automatic diagnosis process that generates results as shown below:



Example of diagnosis results for screens that have more than 2 entry points

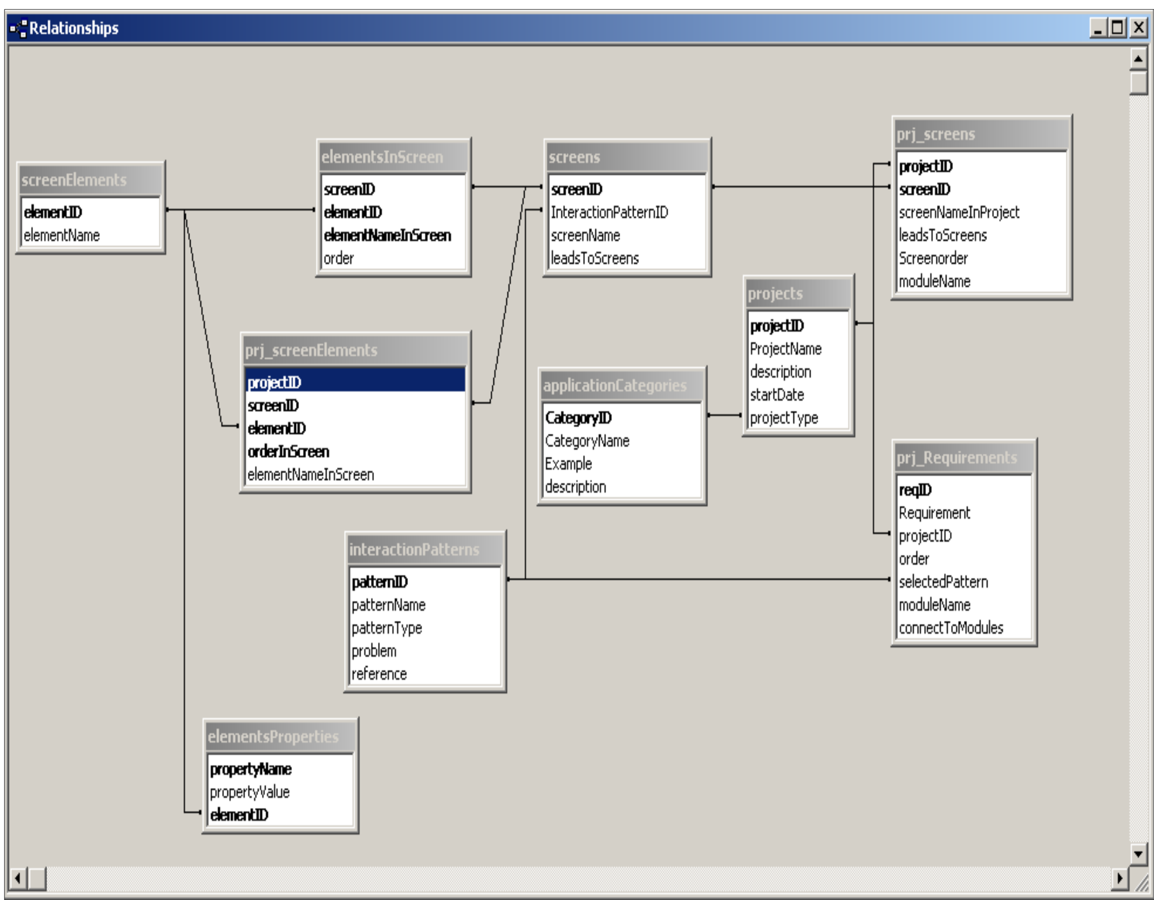
In this example the diagnosis process reveals screens that are directly accessible from more than 2 screens. This is an indication to the designer to reduce the functionality if those screens have a long loading time or alternatively, tell the developers to make it

efficient since it is expected to be requested often. Both these decisions are better handled if identified by the designer before implementation as this process suggests. This capability can also follow the design process, providing immediate feedback on design changes and their implications. The feedback also clarifies why the check has occurred and how to handle the results. A full report can also be generated into a file for later inspection. The main part of the report is given in the appendix A.

Discussion

The 3D application presents a different approach to exposing NPDL to the different persons involved in the navigational design of Web Applications. The design that has been chosen is completely visual, therefore aimed at users without or with little coding experience, but a good grasp of navigational design concepts. The main point behind the development of the 3D application is to show that although the principles of NPDL are based on syntax, it is possible to make it usable and attractive to designers with different levels of skills, regarding programming or grasp of syntactic principles. The underlying database has been normalized in the sense that every type of information is defined once, but used in all relevant tables. We choose not to discuss in depth the principles of the 3D application's architecture, but rather show the functionality with respect to the design of NPDL.

In the next chapter we conclude the project and will present the application within the context of the entire thesis.



Tables and Relationships in the 3D application database

Chapter 7 – Conclusion

In this project we have investigated a method that both standardizes and eases the navigational design process of Web applications for all involved persons. The main approach was to explore navigational design patterns and a language to express both the patterns themselves and the design that use them. In addition, the unification enables the writing of algorithms that analyze the designs before they are implemented.

We began by investigating the domain of navigational design. We started from the functional requirements which are the common ground to the entire Web application design. We continued with the analysis of navigational design patterns, the abstract best-practice solution to recurring problems within the domain. These patterns are often the suggestions of experienced designers who are able to make an abstraction from a solution to a specific problem, into a pattern that matches problems of similar nature. We suggested the unification and connection between the requirements, as problems that need to be addressed by the designer, and the patterns as well-tested solutions. This unification and its need to be conveyed to others (e.g. Designers and developers) have resulted in the exploration of a navigational design language that will enable this unification and standardization.

We went on to define the properties of the language using previous research and original extensions to patterns, such as authorizations and exceptions. Using this definition we have introduced the concepts of our proposed navigational design patterns language

(NPDL). We formalized it and showed examples of known and new patterns as defined by the language. The progression we have taken from there was the introduction if a practical way to implement the language using Standard ML (SML), a functional programming language. We have shown how the concepts of NPDL use the power of SML, such as libraries, simple syntax, standardization and type checking, for the benefit of NPDL requirements. We have demonstrated how to define patterns, use patterns in designs (from libraries) and diagnose the design by writing algorithms in SML.

We went on to construct a top-to-bottom use case based on CampusNet, the DTU university portal. We started by analyzing the requirements, constructing relevant patterns and designing a portion of the application using NPDL. We have shown the capabilities of the language by applying it to the actual application and by creating alternative designs. We analyzed the results using algorithms in order to identify properties of the design and to show the benefits when designing very large systems.

We concluded with the introduction of a visual tool for designers, based on the concepts of NPDL. The tool was constructed using client/server architecture. Beyond providing an alternative way to convey the NPDL design, i.e. Graphical rather than as code, the 3D application demonstrates the important property of the technology-independence of NPDL.

Contributions

The aim of this thesis was to contribute to the research and the applicability of navigational design of web applications. We view the main contribution as the standardization of the means by which

navigational designs are conveyed to others by the designer. By using standard ML and unifying the requirements for such a language from previous research, we have achieved this contribution. We further aimed at defining an applicable and concrete method for defining patterns and libraries that can be used by designers. Again NPDL has been constructed to include these capabilities and we have shown through examples the benefits of the resulting designs.

The problems we have identified in the existing design of CampusNet are described in the discussion in chapter 5. As a reminder they were: that after logging in the user could not change the chosen interface language. The other one was that the course search and selection are in a completely separated application that the registration process. Moreover the registration can be done to each selected course individually and as a complete set of selected courses. By using NPDL and patterns, we have achieved an ability to expose the language identification problem by diagnosis and analysis of the design. For solving the registration design problem we followed these steps:

- Defined a **Catalog with Search** Pattern in NPDL
- Used the **Catalog with Search** and **Basket** patterns in CampusNet's NPDL design of the Course Registration module

We contributed to an improved navigational design for the module. The following user interactions are now possible:

- Presents the courses to the user in a searchable catalog in the application
- Allows a selection of one or more courses into a dedicated list (basket)
- Enables registration within the system to one or more of the

courses in the list at the same time, by selecting the register operation
- The user (student) receives feedback on the registration result of each course. The feedback is similar to the one given in the actual CampusNet, so there is no loss of information

The final contribution we identify is the separation of defining patterns, designing navigational schemes using the patterns and the creation of diagnosis algorithms. This separation is possible due to the introduction of NPDL as a unifying framework. At the same time, NPDL unifies the processes, by connecting functional requirements to concrete designs of the solutions and the ability to analyze corresponding properties. For example, the need to change languages in CampusNet after logging in can be designed and investigated by checking the location and authorizations of the relevant function within the navigational paths. This design property was not investigated in the actual systems. The result is a very partial solution to this problem that was developed at the maintenance stage. An early discovery would have very likely resulted in a correct implementation at the development stage.

Discussion

At this stage we wish to discuss several issues that have been encountered during the writing of this thesis. First, we want to raise **other options** for a basis for NPDL. During the thesis we have introduced symbols for the various parts of the language. However, we have chosen to do this for clarity rather than as a formalization of the syntax. We chose to follow the path of using SML as a basis for defining and extending the language, but we could have created a self-defined syntax. The reason for not doing

this is that we did not find it useful to make a new syntax, but use the well-defined and tested framework of SML. This choice enhances, in our opinion, the robustness of NPDL and shortens the learning curve associated with introduction of new languages. We have related the functional Requirements to the design. The navigational design process naturally continues to the implementation phase. We have used SML also for the reason that the output of the interpreter is well defined. This fact makes it possible to convert the output of an NPDL to numerous other formats, such as XML or HTML and generate input to other applications. These applications may further process the design (e.g. Validations, type checking or simulations) or alternatively generate the HTML code for the entire application, based on the design. This enables **integration** between the design phase and the implementation phase, a link which can make the entire development process more efficient, cost-effective and at the same time less error-prone and time consuming.

In considering other approaches to the same goals we may have chosen an object-oriented approach, i.e. creating navigational design patterns as Java classes. Although the main considerations for not doing this were the significantly increased syntactic complexity, that would have been imposed on the designer and on the other hand, the need to handle instantiations of patterns using abstract structure, a process that is more cumbersome in Java. Other object-oriented languages were dismissed for the same reasons. Taking a graphical approach as a platform could have been successful, but would have meant a more technical and practical thesis. We would have had to develop a graphical interface that can show the navigational design of the Web application in various levels of granularity. One can imagine the ability to zoom into a graphical representation of a module in order

to view the sub-modules and screens that it is comprised of and so on. This approach, though visually appealing, deprives the solution from being compact and clear for large applications, such as enterprise resource planning (ERP) systems.

Conclusions

During the research and the writing of this thesis, I have learned a great deal about the need for patterns within the domain of navigational design. There has been considerable work done in attempting to define the attributes of patterns and specific patterns. The need for a language that supports this process has been discussed as well. However, I have found a lack of concrete solutions to both of these issues in a way that takes into account all the aspects of the process as well as those that connect it to its predecessor (Functional Requirements) and the successor, i.e. the implementation of the Web applications. I have also noticed the lack of reference to a key issue, i.e. the handling of exceptions and an important issue of authorizations.

The set of goals that have been defined in the beginning of the project have been successfully accomplished, although initially the way in which they would be achieved was completely unknown. The compact method of describing designs and using navigational patterns has been achieved by defining the language NPDL. The use of SML as basis provides standardizations and ability to define and publish patterns libraries. We have further shown that this method enables the creation and execution of diagnostic algorithm hat can check various design properties. The resulting designs, due to the clarity and testing, are expected to be more robust. Their standard defined format can be used by other systems as well. Therefore we think that the project has been a success and provides

a basis that complies with the broad set of objectives, as well as being an extensible and scalable platform for further developments. Some of those possible enhancements are discussed in the next section.

Further Work

In this final section of the thesis we will shortly consider possible directions for further work. One important extension would be the introduction of more types of relationships between patterns. We discussed in depth the hierarchical connection, but there are others like IF...THEN and WHILE connections, implying that the construction of complex patterns can be based on conditions that are satisfied within some screens.

In the short-term, the following topics would need to be addressed:

- Addition of screen-elements
- Implementation of exceptions in the SML construction, based on the concept that has been introduced
- Formalization of the NPDL syntax, based on a symbolic representation or on SML syntax. Both have been informally introduced throughout the project

In more long-term perspective, the following topics would be interesting to consider:

- A full-blown library of patterns in NPDL
- A search mechanism based on the NPDL to identify patterns based on search criteria
- Writing a (semi-) automatic algorithm for matching the application's functional requirements and available patterns for enhanced and more accurate pattern selection process.

As presented in chapter 6, we have shown the development of a visual tool for NPDL pattern and Web application designers. We see the user-interface, the underlying framework (NPDL implementation and concepts) and the database as being close to a commercial product within the market of Web Applications navigational design. The required additions would be improvement of the screens and handling of exceptions. Otherwise the product handles all the other properties of the language and provides a viable tool for designer, as presented.

Bibliography

1. [AG1] M.A.K. Akanda and D.M. Germán. **A Component-Oriented Framework for the Implementation of Navigational Design Patterns.** *Proceedings of the International Conference on Web engineering, 445-448, 2003*
2. [AIS1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel. **A Pattern Language**. *Oxford University Press, NY* 1977
3. [AVL1] A. van Lamsweerde, **Requirements Engineering in the Year 00: A Research Perspective.** *Proceedings of the 22nd International Conference on software engineering, 5-19, 2000*
4. [BCM1] D. Bonura, R. Culmone and E. Merelli. **Patterns of Web Applications.** *Proceedings of the 14th international conference on Software engineering and knowledge engineering, 739-746, 2002*
5. *[DDMP1]* E. Di Sciascio, F.M. Donini, M. Mongiello and G. Piscetelli. **AnWeb: a System for Automation Support to Web Application Verification.** *Proceedings of the 14th International Conference on Software engineering and knowledge engineering, 609-616, 2002. ISBN 1-58113-556-4*
6. *[DFAM1]* A.. Dearden, J. Finlay, E. Allger and B. McManus. **Using Pattern Languages in Participatory Design.** *In Binder, T., Gregory, J. & Wagner, I (Eds.) PDC 2002, Proceedings of the Participatory Design Conference. CPSR, Palo Alto, CA.,2002. ISBN 0 9667818-2-1. pp. 104 - 113*
7. [GC1] D.M. Germán and D.D. Cowan. **Towards a Unified Catalog of Hypermedia Design Patterns.** *Proceedings of the 33rd Hawaii International Conference on System Sciences, 2000*
8. [GHJV] E. Gamma, R. Helm, R. Johnson and J. Vlissides. **Design Patterns. Elements of reusable object-oriented**

software. Addison Wesley, 1995

9. [GSV1] N. Güell and D. Schwabe and P. Vilain. **Modeling Interactions and Navigation in Web Applications.** *Second International Workshop on the World Wide Web and Conceptual Modeling, (WCM 2000), EE 115-127*
10. [HAN] M. R. Hansen and H. Richel. **Introduction to Programming Using SML**. ISBN 0-201-39820-6, Addison-Wesley, 1999.
11. [HK1] J. Hannemann and G. Kiczales. **Design Pattern Implementation in Java and aspectJ.** *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, systems, languages and applications,p. 161-173, 2002*
12. [KR1] G. Kappel, W. Retschitzgger et al. **Ubiquitous Web Application Development – A Framework for Understanding.** *The sixth Multiconference on Systematics, Cybernetics and Informatics SCI2002, 2002*
13. [KRS1] G. Kappel, W. Retschitzgger and W. Schwinger. **Modeling Customizable Web Applications – A Requirement's Perspective.** *Kyoto International Conference on Digital Libraries, 2000*
14. [KUH1] T. Kühne. **A Functional Pattern System for Object Oriented Design.** Ph.D. Thesis *ISBN 3-86064-770-9, 1999*
15. [MB1] M. Bernstein, **Patterns of Hypertext.** *Proceedings of the ninth ACM conference on Hypertext and hypermedia : links, objects, time and space---structure in hypermedia systems, 21-29, 1998*
16. *[MHG1] D*. Maplesden, J. Hosking and J. Grundy, **Design Pattern Modelling and Instantiation using DPML .** *Proceedings of the fortieth International Conference on Toll specific: objects of Internet, mobile and embedded applications ,Volume 10 p. 3-11, 2002*

17. [NNK1] M. Nanard, J. Nanard and P. Kahn. **Pushing Reuse in Hypermedia Design: Golden Rules, Design Patterns and Constructive Templates**. *Proceedings of the Ninth ACM Conference on Hypertext and hypermedia: links, objects, time and space – structures in Hypermedia Systems, 11-20, 1998*
18. [RSL1] G. Rossi, D. Schwabe and F. Lyardet. **Improving Web Information System with Navigational Patterns**. *Computer Networks 31 (1999), 1667-1678*
19. [RSL2] D. Schwabe, G. Rossi and F. Lyardet. **Web Application Models are More Than Conceptual Models**. *ER Workshops 239-253, 1999*
20. [RSL3] D. Schwabe, G. Rossi and F. Lyardet. **Abstraction and Reuse Mechanisms in Web Application Models**. *ER Workshops 76-88, 2000*
21. [SJFS1] *D. Sinnig, H. Javahery, P. Forbrig and A. Seffah.* **A Complicity of Model-Based Approaches and Patterns for UI Engineering** . *In Proceedings of BIR, p. 120-131, SHAKER, 2003*
22. [SREL1] D. Schwabe, G. Rossi, L. Esmeraldo, and F. Lyardet. **Engineering Web Applications for Reuse**. *IEEE Multimedia, 8(1):20-31, 2001*
23. [SSBZ1] J. Schümmer, C. Schuckmann, L.M. Bibbó and J.J. Zapico. **Collaborative Hypermedia Design Patterns in OOHDM**. *Second Workshop in Hypermedia Development: Design Patterns in Hypermedia, 1999*
24. [WV1] M. van Welie, G.C. van der Veer. **Pattern Languages in Interaction Design: Structure and Organization.** *Proceedings of Interact '03, Zürich, Eds: Rauterberg, Menozzi, Wesson, p527-534, ISBN 1-58603-363-8, IOS Press, Amsterdam, The Netherlands, 2003*

Appendix A - Code Samples

In this appendix we include code samples that did not fit in the main thesis text. The first code is for printing the hierarchical type patterns defined as a tree.

```
datatype Application =
   Empty
            | Leaf of scrset.set
            | Node of scrset.set * Application list;

fun insertSon (x, Empty) = Leaf(x)
            | insertSon (x, Leaf(y)) = Node (y,[Leaf(x)])
            | insertSon (x, Node(y, app))   = Node(y,app@[Leaf(x)]);

fun   insertNode (x, Empty) = x
           | insertNode (x, Leaf(y)) = Node(y, [x])
           | insertNode (x, Node(y, app)) = Node(y,app@[x]);

fun   scrsetToString [] prefix =  ""
           | scrsetToString (sc::scset) prefix = prefix^ "(" ^ Int.toString (#1 sc) ^ "," ^ (#4 sc) ^ "," ^ (#2 sc) ^
")" ^ "\n" ^ prefix ^ (scrsetToString scset prefix)

fun treeToString (Empty, prefix) = prefix ^ "[]"
  | treeToString (Leaf(x), prefix) = prefix ^ (scrsetToString (scrset.toList(x)) prefix)
  | treeToString (Node(x, []), prefix) = treeToString (Leaf(x), prefix)
  | treeToString (Node(x, app), prefix) = treeToString (Leaf(x), prefix)^ "\n"  ^ treeToString (List.hd(app),
prefix ^ "  " )   ^ treeToString (Node(scrset.empty, List.tl(app)),prefix) ;
```

Code for printing hierarchical type patterns

Screen Components Code

We present the code for the screen elements in NPDL:

```
datatype inptypes = String | Numeric | Alphanumeric | Date ;

structure Link =
struct
```

```
    val myType = "link"
    type Name = string
    type Target = int  (*genericscreen*)
    type NewWin = bool
end

structure Button =
struct
    val myType = "button"
    type Name = string
    type Function = defaultFunction.Function
    type clickableOnEntry = bool
end

structure InputBox =
struct
   val myType = "inputbox"
    type Name = string
    type InputType = inptypes
    type isMasked = bool
    fun newInputBox Name InputType isMasked = (Name,InputType,isMasked)
end

structure CheckBox =
struct
    val myType = "checkbox"
    type Name = string
    type Function = defaultFunction.Function
    type checkedOnEntry = bool
end

structure SelectSingleOption =
struct
    val myType = "selectsingleoption"
    type Name = string
    type DataSource = string
end

structure SelectMultipleOptions =
struct
    val myType = "selectmultipleoption"
    type Name = string
    type DataSource = string
end

structure DataTable =
struct
    val myType = "datatable"
    type Name = string
    type DataSource = string
end
```

```
structure DisplayText =
struct
        val myType = "displaytext"
   type Name = string
   type DataSource = string
end
```

```
datatype screenElement = link of (Link.Name * Link.Target * Link.NewWin) |
        button of (Button.Name * Button.Function * Button.clickableOnEntry) |
        checkbox of (CheckBox.Name * CheckBox.Function * CheckBox.checkedOnEntry) |
        singleOption of (SelectSingleOption.Name * SelectSingleOption.DataSource) |
        multipleOptions of (SelectMultipleOptions.Name * SelectMultipleOptions.DataSource) |
        dataTable of (DataTable.Name * DataTable.DataSource) |
        displayText of (DisplayText.Name * DisplayText.DataSource) |
        inputbox of (InputBox.Name * InputBox.InputType * InputBox.isMasked);
```

Complete Element Definitions and Screen Elements as defined in NPDL

CampusNet Design in NPDL

We have designed a certain portion of CampusNet using NPDL in order to demonstrate a practical example of using the language. Following the NPDL code that is the design of the portion in Campusnet:

```
load "IntSet";
load "Int";
load "Real";
load "Date";
use "setLib.sml";
use "htmlLib.sml";
use "tree.sml";
use "patternsLib.sml";
use "diagnosisLib.sml";

(* Campusnet Functional Requirements Definition *)
val fr = FunctionalRequirements.insert((1,"register to courses"),
                FunctionalRequirements.insert((2,"view student grades"),
                FunctionalRequirements.insert((3,"view course participants list"),
                FunctionalRequirements.insert((5,"secure login to the system"),
                FunctionalRequirements.insert((6,"allow viewing course-participant details"),
                FunctionalRequirements.insert((4,"send messages to course participants"),
                FunctionalRequirements.insert((7,"track course related activities in course and personal
calendar"),
                FunctionalRequirements.insert((8,"the system must enable adding, viewing and
searching for courses"),
                FunctionalRequirements.insert((10,"allows users to access main modules, such as
messages and email from all screens"),FunctionalRequirements.empty)
```

```
                                ))))))));

(*************************' Campusnet Modules Design *****************************)
val  loginModuleName = "campusnet login";
        val  myCalendarModuleName = "MY Calendar";
        val  courseModuleName = "course registration module";
                val  courseCatalogModuleName = "course catalog";
        val  groupModuleName = "CN Groups (courses, DTU, user)";
                val  calendarModuleName = "course activities calendar";
                val  courseParticipantModuleName = "course participants";

(* campusnet login *)
(* campusnet user - example of overriding a defaultUser structure if required *)
structure CampusnetUser : ApplicationUserSig =
struct
                type Id = int
    type Password = string
    type authorizationGroups = AuthGroupIDset.set
    type user = (Id * Password * authorizationGroups)
    fun newUser id = (id, "****",AuthGroupIDset.empty)
    fun hasAuthorization (id,p,ag) a = if AuthGroupIDset.member(a,ag) then true else false
          fun addAuthorization (id,p,ag) a = (id,p,AuthGroupIDset.insert(a,ag))
          fun deleteAuthorization (id,p,ag) a = (id, p, AuthGroupIDset.delete(a,ag))
    fun printUser (id,p,ag) = id::AuthGroupIDset.toList(ag)
    fun getLogin (id,p,ag) = (id, p)
end

structure loginProperties : propertiesDesignSig =
struct
                val  moduleName = loginModuleName;
                val  fulfillsRequirements = (reqIDset.linsert([5],reqIDset.empty));
                val  reachableScreens = (scrIDset.insert((200,calendarModuleName),

scrIDset.insert((200,myCalendarModuleName),

scrIDset.insert((100,courseModuleName),scrIDset.empty)
                                )));
end

structure loginModuleDesign = loginFct(structure Usr=CampusnetUser; structure GD=loginProperties);

(* campusnet  course activities in calendar *)
structure CampusnetActivity = defaultActivity;
structure ActivityCalendarProperties : propertiesDesignSig =
struct
                val  moduleName = calendarModuleName;
                val  fulfillsRequirements = (reqIDset.linsert([7],reqIDset.empty));
    val  reachableScreens = (scrIDset.empty);
end

structure calendarModuleDesign = calendarFct(structure Act=CampusnetActivity; structure
GD=ActivityCalendarProperties);
```

```
(* campusnet  personal activities in calendar *)
structure PersonalActivityCalendarProperties : propertiesDesignSig =
struct
       val  moduleName = myCalendarModuleName;
       val  fulfillsRequirements = (reqIDset.linsert([7],reqIDset.empty));
       val  reachableScreens = (scrIDset.insert((2,loginModuleName),

scrIDset.insert((100,courseModuleName),scrIDset.empty)
                                    ));
end

structure myCalendarModuleDesign = calendarFct(structure Act=CampusnetActivity; structure
GD=PersonalActivityCalendarProperties);


(* campusnet course Registration Module *)
structure CourseItem = defaultItem;
structure CourseProperties : propertiesDesignSig =
struct
                        val  moduleName = courseModuleName;
                        val  fulfillsRequirements = (reqIDset.linsert([1],reqIDset.empty));
       val  reachableScreens = (scrIDset.insert((2,loginModuleName),

scrIDset.insert((300,courseCatalogModuleName),

scrIDset.insert((200,calendarModuleName),scrIDset.empty)
                                    )));
end

structure courseModuleDesign = basketFct(structure Itm=CourseItem; structure GD=CourseProperties);

val courseModulefl = courseModuleDesign.FuncList [
                                                            defaultFunction.newFunction "register" [("int","semester
season"),("int","semester year"),("int","student id")] [(1,"approved"),(2,"full"),(3,"not given this
semester")],
                                                            defaultFunction.newFunction "print" [("int","printer
ID"),("int","number of copies")] [(1,"course info printed"),(2,"course info unavailable"),(3,"printer not
responding")],
                                                            defaultFunction.newFunction "unregister" [("int","student
ID")] [(1,"unregister successful"),(2,"unregister failed")]
                                                ];

(* campusnet  course participants *)
structure CourseParticipant = defaultItem;
structure CourseParticipantProperties : propertiesDesignSig =
struct
       val  moduleName = courseParticipantModuleName;
       val  fulfillsRequirements = (reqIDset.linsert([3,6],reqIDset.empty));
       val  reachableScreens = (scrIDset.empty);
end
```

```
structure courseParticipantModuleDesign = catalogFct(structure Itm=CourseParticipant;structure
GD=CourseParticipantProperties);


(* campusnet  course catalog *)
structure CourseCatalogProperties : propertiesDesignSig =
struct
       val  moduleName = courseCatalogModuleName;
       val  fulfillsRequirements = (reqIDset.linsert([8],reqIDset.empty));
       val  reachableScreens =
(scrIDset.linsert([(2,loginModuleName),(100,courseModuleName)],scrIDset.empty)
                                                                           );
end

structure courseCatalogModuleDesign = searchCatalogFct(structure Itm=CourseItem;structure
GD=CourseCatalogProperties);

(************************* AUTHORIZATION DEFINITIONS
****************************)

val title = "%%%%%%%%%%%%%%%%%%%%%%%%%%%%% authorization definitions
%%%%%%%%%%%%%%%";
val adminasc = scrIDset.insert((10000,"all"), scrIDset.empty);
val adminfsc = scrIDset.empty;
defaultAuthGroup.newAuthGroup 1 "admin" adminasc adminfsc;
val studasc =
scrIDset.linsert([(1,loginModuleName),(2,loginModuleName),(3,loginModuleName),(100,courseModuleN
ame),(101,courseModuleName),(102,courseModuleName)], scrIDset.empty);
val studfsc = scrIDset.insert((200,calendarModuleName), scrIDset.empty);
defaultAuthGroup.newAuthGroup 2 "students" studasc studfsc;
val lecasc =
scrIDset.linsert([(1,loginModuleName),(200,calendarModuleName),(200,myCalendarModuleName)],
scrIDset.empty);
val lecfsc = scrIDset.empty;
defaultAuthGroup.newAuthGroup 3 "lecturers" lecasc lecfsc;

(**************************** Campusnet Application Design
*****************************)
structure groupModuleDesign =
struct
          structure Generic: genericDesignSig =
   struct
              val screens = scrset.linsert([defaultScreen.newScreen 10000 "Group Welcome"
[(displayText("welcome...",""),1.0),(link("search again",400,false),2.0)] groupModuleName],scrset.empty);
                     val numOfScreens = scrset.setsize(screens);
   end
   structure Properties =
   struct
     val  moduleName = groupModuleName;
     val  fulfillsRequirements = reqIDset.empty;
     val  reachableScreens =
```

```
(scrIDset.linsert([(2,loginModuleName),(100,courseModuleName)],scrIDset.empty));
        end
end

structure campusnetDesign1 =
struct
   structure LOG = loginModuleDesign
   structure CAT = courseCatalogModuleDesign
   structure CAL = myCalendarModuleDesign
   structure REG = courseModuleDesign
   structure GRP = groupModuleDesign
   structure Generic: genericDesignSig =
   struct
            val screens =
scrset.sinsert(LOG.Generic.screens,scrset.sinsert(GRP.Generic.screens,scrset.sinsert(REG.Generic.screens,
(scrset.sinsert(CAT.Generic.screens,CAL.Generic.screens)))));
                  val numOfScreens = scrset.setsize(screens);
   end
end

(* designing the entire application as an architectural design pattern *)
val campusnetDesign = insertSon(myCalendarModuleDesign.Generic.screens,
Leaf(loginModuleDesign.Generic.screens));
val campusnetDesign =
insertNode(Node(courseModuleDesign.Generic.screens,[Leaf(courseCatalogModuleDesign.Generic.screen
s)]),campusnetDesign);
val campusnetDesign = insertNode(Node(groupModuleDesign.Generic.screens,
[Leaf(calendarModuleDesign.Generic.screens),Leaf(courseParticipantModuleDesign.Generic.screens),Leaf
(courseParticipantModuleDesign.Generic.screens)]), campusnetDesign);
```

campusnet design in NPDL

CampusNet Design Diagnosis Report

We have chosen to display a part of the diagnosis report generated by the 3D application. This report is based on the design of CampusNet within the Application.

```
THIS IS A DESCRIPTION OF THE DESIGN STRUCTURE
GENERATED AUTOMATICALLY BY 3D APPLICATION ON 6/15/2004 10:53:34
*********************************************************************
*************
Project Name: campusnet   Type: 2
Desciption: A system for teachers and students at DTU
*********************************************************************
*************
-------------------------------------------------------------
```

Screen name: compose message To participants

Module name: course management

Screen Order In Module: 1

The following elements are defined for this screen:
  1. Group Identifier (inputField)
  2. Priority (select1FromMany)
  4. Expires (inputField)
  5. Heading (inputField)
  6. Text (inputField)
  7. Link (inputField)
  8. Send (button)

Leads to the following screens: View Course messages,

The following screens lead to this screen:
  1. View Course messages

Screen name: View Course messages

Module name: course management

Screen Order In Module: 2

The following elements are defined for this screen:
  1. Course Messages (DataTable)
  2. Compose Message (button)

Leads to the following screens: compose message To participants,

The following screens lead to this screen:
  1. compose message To participants
  2. bar

-
Screen name: Register to Selected Courses

Module name: course registration

Screen Order In Module: 1

The following elements are defined for this screen:
  1. selected courses (link)
  3.1. REGISTER (button)
  3.2. PRINT INFORMATION (button)

Leads to the following screens: Enter Semester Details,

The following screens lead to this screen:
1. campusnet main screen
2. Registration Confirmations
3. list of grades
4. bar

Screen name: Enter Semester Details

Module name: course registration

Screen Order In Module: 2

NO ELEMENTS ARE DEFINED FOR THIS SCREEN

Leads to the following screens: Registration Confirmations,

The following screens lead to this screen:
1. Register to Selected Courses

Screen name: Registration Confirmations

Module name: course registration

Screen Order In Module: 3

The following elements are defined for this screen:
1. registration feedback (displayText)

Leads to the following screens: Register to Selected Courses,

The following screens lead to this screen:
1. Enter Semester Details

Screen name: list of grades

Module name: Grades

Screen Order In Module: 1

NO ELEMENTS ARE DEFINED FOR THIS SCREEN

Leads to the following screens: enter grades, main screen, Register to Selected Courses, view grades,

The following screens lead to this screen:
  1. campusnet main screen
  2. view grades
  3. enter grades

Screen name: view grades

Module name: Grades

Screen Order In Module: 2.1

NO ELEMENTS ARE DEFINED FOR THIS SCREEN

Leads to the following screens: list of grades,

The following screens lead to this screen:
  1. list of grades
  2. enter grades
  3. bar

Screen name: enter grades

Module name: Grades

Screen Order In Module: 2.2

NO ELEMENTS ARE DEFINED FOR THIS SCREEN

Leads to the following screens: list of grades, view grades,

The following screens lead to this screen:
  1. list of grades

Screen name: login

Module name: logging in

Screen Order In Module: 1

The following elements are defined for this screen:
  1. student id (inputField)
  2. password (inputField)
  3. logon (button)
  4. remember me? (2StateIndicator)

Leads to the following screens: main screen,

NO SCREENS LEAD TO THIS SCREEN

Screen name: campusnet main screen

Module name: logging in

Screen Order In Module: 2

The following elements are defined for this screen:
  1. messages (DynamicContent)
  2. news (DynamicContent)

Leads to the following screens: list of grades, Register to Selected Courses,

NO SCREENS LEAD TO THIS SCREEN

Screen name: bar

Module name: main menu bar

Screen Order In Module: 1

The following elements are defined for this screen:
  1. my messages (link)
  2. my calendar (link)
  3. Grades (link)
  4. history (link)
  5. Address Book (link)
  6. Course Registration (link)
  7. email (linkToNewWindow)

Leads to the following screens: Register to Selected Courses, View Course messages, view grades,

NO SCREENS LEAD TO THIS SCREEN

****
Project Name: salesforce.com   Type: 5
Desciption: online CRM system
****

Screen name: Event Details

Module name: sf calendar

Screen Order In Module: 1

The following elements are defined for this screen:
  1. start time (displayText)
  2. meeting topic (displayText)
  3. save (button)

Leads to the following screens: calendar Overview,

The following screens lead to this screen:
  1. calendar Overview

Screen name: calendar Overview

Module name: sf calendar

Screen Order In Module: 1

The following elements are defined for this screen:
  1. all activities by date (DynamicContent)
  2. event details (button)

Leads to the following screens: Event Details,

The following screens lead to this screen:
  1. Event Details