Dienstag, November 20, 2001 08:46

Final version for RE Journal

Round-Trip Prototyping based on Integrated Functional and User Interface Requirements Specifications

Andreas Homrighausen, Hans-Werner Six, Mario Winter
University of Hagen, Dept. of CS, Hagen, Germany

Requirements engineering in the new millennium is facing an increasing diversity of computerized devices comprising an increasing diversity of interaction styles for an increasing diversity of user groups. Thus the incorporation of user interface requirements into software requirements specifications becomes more and more mandatory. Validating these requirements specifications with hand made, throw away prototypes is not only expensive, but also bears the danger that validation results are not accurately fed back into the requirements specification. In this paper, we propose an enhancement of the requirements specification method SCORES *for an explicit capturing of user interface requirements. The advantages of the approach are threefold. First, the user interface requirements specification is UML-compliant and integrated into the functional requirements specification. Second, prototypes for validation purposes can semi-automatically be generated. Third, the model based generation of prototypes allows for "round-trip prototyping" such that manual changes of the prototype during the validation process are automatically fed back into the requirements specification.*

Keywords: Requirements engineering; User Interface Requirements; UML; Prototyping; Round-Trip Prototyping; Validation

1 Introduction

Due to the "ubiquitous computing" paradigm, the new millennium opens with an increasing diversity of computerized devices comprising an increasing diversity of interaction styles for an increasing diversity of user groups [26]. For example, modern eBusiness domains demand for application families with shared functional cores but vastly different device and user interface requirements. The diversity of devices ranges from narrowband devices, including cellular phones and pagers, up to workstations with high resolution graphic displays, while the diversity of interaction styles ranges from the deck/card organizational metaphor up to high sophisticated multi-window graphical user interfaces. Finally, user groups range from incidental to professional ones. Hence, as already pointed out in [16], a meaningful software requirements specification (SRS) in such application domains must sufficiently take into account user interface requirements.

For practical use, it is important that abstraction level and modelling constructs of functional and user interface requirements specifications match well such that they can be composed into an integrated overall specification. Moreover, like any suitable SRS such an entire specification must support validation, verification, and test processes.

To achieve these goals, we have enhanced SCORES [21], an approach for a UML-based description of the functional, behavioural, and structural system requirements, by essential elements of FLUID [20], a specification method for the requirements of direct manipulation user interfaces. We call the resulting integrated SRS framework SCORES+.

The original SCORES method provides a refinement of use cases (actually of the use case behaviour) to get a proper coupling of use cases and domain class model which allows for consistency and completeness checks of the multi model specifications. Concerning the validation process, SCORES provides inspections and walkthroughs of (business) scenarios [17][19] which are recommended validation means [4][25][38].

Using SCORES in a couple of applications with different devices, interaction styles, and user groups has revealed some weaknesses of the validation procedure. First, although we provided domain specific pictograms and screen mock-ups, sometimes the imagination of the users was overstrained and a more tangible representation

Correspondence and offprint requests to: Mario Winter, FernUniversität Hagen, Prakt. Inf. III / IZ, Feithstr. 142, D-58084 Hagen, Germany. Email: mario.winter@fernuni-hagen.de

seemed desirable. Second, wrong or missing requirements, which were detected during the validation and materialized by changed pictures or mock-ups, did not always trigger the necessary changes of the requirements specifications.

Prototyping is usually recommended as a more appropriate validation technique in such a context. In requirements engineering for information systems, often exploratory presentation prototypes and sometimes functional prototypes are used to validate the SRS [2][11]. While presentation prototypes illustrate how an application may solve given requirements from a user interface point of view, functional prototypes implement important parts of both, the user interface and the functionality of the application.

Unfortunately, developing a meaningful prototype is a hard and expensive work to do [32]. A purely generative approach of building prototypes needs a detailed specification and does not allow for an easy change or extension of prototypes during validation sessions. Hand made throw away presentation prototypes are also costly to develop and bear the danger that changes of the prototype are not consistently fed back into the SRS. As pointed out by Ravid and Berry, the question “When the prototype and the SRS disagree, which do one believe?“ often arises [30].

To tackle the problems mentioned above, we extend SCORES by FLUID user interface modelling elements in a UML-compliant way. FLUID captures the static structure of the user interface (a basic description of scenes, e.g. windows, and their relationships), the basic dynamic behaviour (e.g. the navigation structure), and (parts of) the domain objects to be presented [20]. Before getting FLUID elements integrated into SCORES we had to refine them for this purpose.

The resulting integrated method SCORES+ allows not only for a combined specification of functional and user interface requirements but also — mainly due to FLUID — for a cheap, semi-automatic construction of exploratory, horizontal (presentation) prototypes for different devices and interaction styles. Moreover, because of the traceability of the prototype construction process *“round-trip prototyping*“ is carried into effect and the requirements specification can automatically be reconciled with manual changes of the prototype.

The paper is structured as follows. Section 2 briefly reviews the specification of functional requirements with SCORES, while section 3 is devoted to the refinement of FLUID elements and their integration into SCORES. Section 4 deals with prototyping and explains the round-trip prototyping process. Section 5 summarizes our experiences with the prototype based validation. Section 6 concerns the related work and section 7 concludes the paper.

2 Functional Requirements

In this section we briefly review the basic modelling elements of SCORES [21], namely actors and use cases, activity graphs, and the domain class model.

2.1. Actors, Use Cases, and Activity Graphs

According to the UML [29], *actors* characterize roles played by external objects which interact with the system as part of a coherent work unit (a use case). A *use case* describes a high level task which should be related to a particular goal [28] of its participating actors. The functional profile of a use case comprises usage frequency, criticality, and some risk factors (R_T, how difficult to implement, R_B, publicity, sales arguments, and R_P, how many dependencies). Additionally we add to each use case the minimal interaction style requirements like desktop/ WIMP, navigational (HTML), or cardbox (WML).

The UML proposes the *activity graph* concept for the specification of processes involving one or more classifiers [29]. In [21] we extend activity graphs to meet the needs for a suitable modelling of use case behaviour. We describe actions within activity graphs more precisely by introducing three stereotypes. An action stereotyped «*contextual action*» is performed by an actor without the help of the system. An action stereotyped «*interaction*» is supported or carried out by the system. The «*macro action*» stereotype reflects the «*include*» and «*extend*» relationships between use cases in the context of activity graphs by "calling" or "activating" the activity graph of the associated use case. The stereotype «*actor in action*» refers to an (anonymous) instance or “role“ [6] of an actor involved in an action.

2.2. Domain Class Model

The domain class model in the sense of Jacobson et al. [17] comprises the most important types of objects which represent "things" that exist or events that occur in the context in which the system works. In SCORES [21], for each use case and each interaction in the activity graphs we determine a so-called *class scope* denoting the set of (domain) classes involved in its execution, i.e. classes with instances being created, retrieved, activated, updated, or deleted. Furthermore we balance activity graphs and the class model such that for each «*interaction*» in the activity graphs a so-called *root class* exists that provides a *root operation* accomplishing the «*interaction*» in the class model.

Running Example: We take WEBADMIN, a virtual university [7] component for the web-based administration of events at the FernUniversität Hagen, the German distance teaching university, as running example. WEBADMIN covers the scheduling of, registration for, and accomplishment of written examinations and seminars.

Students can attend written examinations at two different dates at some 10 different sites in Germany, Austria, and Switzerland. In the left part of Fig. 1, the use case diagram for the registration functionality of WEBADMIN is shown. We identify the actor Student and the use cases

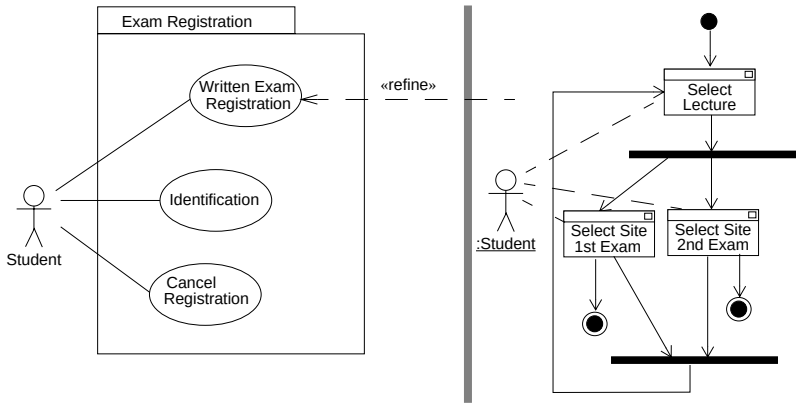

Fig. 1. Use case diagram and activity graph for use case Written Exam Registration

Identification, Written Exam Registration, and Cancel Registration. Fig. 2 provides a textual description of the use case Written Exam Registration. Fig. 3 depicts some domain classes. The class scope of the use case WrittenExamRegistration consists of the classes Event, Lecture, WrittenExam, and Registration.

The activity graph in the right part of Fig. 1 refines use case Written Exam Registration. A student can (repeatedly) select a lecture and registrate for the first and/or the second examination of this lecture at a particular site. The interactions Select Site 1st Exam and Select Site 2nd Exam invoke their corresponding root operation registerTo() defined in the class Event.

Use Case Written Exam Registration **in Model** WebAdmin
Actors: Student, Supervisor
Usage Frequency: high
Criticality: low
Risk: {Rt=medium, Rb=high, Rp=high}
Minimal Interaction Style: cardbox
Pre-Condition: Student identified **and** exam sites exist
Description: A student registers for the offered written exams by selecting one of the certified sites.
Extension Point Special Case Student
Post-Condition: 1st exam registration **or** 2nd exam registration **or** special case registration conducted
End Written Exam Registration

Fig. 2. Textual descriptions of use case Written Exam Registration

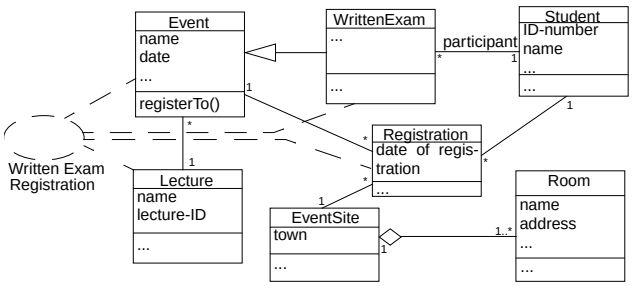

Fig. 3. WEBADMIN Domain class diagram

3 User Interface Requirements

As explained in the introduction, a suitable SRS should include the basic characteristics of the user interface, among them the static structure, the basic dynamic behaviour, and usability aspects. It may simply comprise a list of do's and don'ts on how the system will appear to the user.

To specify user interface requirements more precisely and to support the convenient construction of presentation prototypes for validation purposes, we exploit FLUID [20]. As a user interface analysis (UIA) method, FLUID captures only the most important decisions concerning the user interface and specifies neither the concrete interaction style nor the detailed GUI design.

FLUID as introduced in [20] does not address a full fledged SRS as needed in the application context described in the introduction of this paper. It also does not use the UML as modelling language because the UML was not at hand at that time. Therefore, we first refined the FLUID elements to meet the abstraction level of SCORES elements and made them UML compliant. Then we integrated the refined FLUID elements into SCORES by adding them to use cases and activity graphs appropriately.

The resulting UIA elements provide modelling concepts aiming at complex user interfaces with a broad range of display and interaction styles. They are based on the assumptions that a direct manipulation user interface is composed of building blocks, like e.g. windows and sub-windows, and that domain objects (more precisely: their attributes and associations) appear inside of such building blocks where they can be viewed, selected and directly manipulated by the user.

3.1. UIA static structure: Scenes and Class Views

The fundamental UIA elements are scenes comprising class views and user operations. Scenes together with their relationships and class views determine the *static structure* of the user interface. The scene concept provides for a

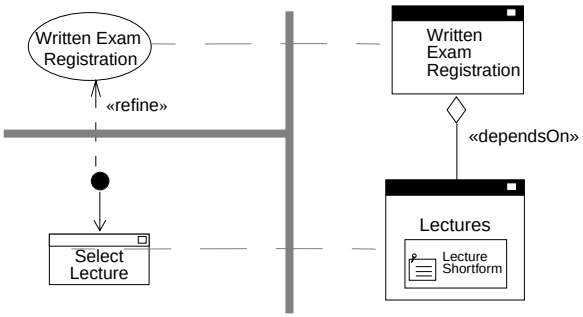

Fig. 4. Use case Written Exam Registration with partial activity graph, Scenes and a class view

homogeneous decomposition of a user interface into its main building blocks. A scene may later be implemented as a window or sub-window, a deck of cards, or a similar user interface feature. However, these implementation details are beyond the focus of requirements engineering. We come back to these details in section 4 where we discuss the semi-automatic construction of prototypes.

A scene is modelled by a UIA class stereotyped «*scene*» and complements a use case or an «*interaction*» of an activity graph. In the first case, the scene is called the *start scene* of the use case. A scene can be characterized as *multiple* (there may exist multiple instances of the scene) and/or *modal* (all other scenes are locked until the according task is completed).

An n-ary association relationship stereotyped «*mutually exclusive*» denotes that each associated scene is modal with respect to only the associated scenes rather than to all scenes. At any time, at most one of the associated scenes is visible and available to the user.

An aggregation relationship stereotyped «*dependsOn*» between two scenes S and T denotes that scene S only exists or is visible if scene T exists or is visible. For example, in many cases there exists a «*dependsOn*» relationship between the start scene of a use case and the scenes of the interactions in the according activity graph.

A scene may comprise class views. A *class view* is modelled by a UIA class stereotyped «*class view*». It is an abstract presentation of instances and associations of a domain class involved in the according user activity (i.e. contained in the class scope). A scene with no class view reflects no domain object information and often serves only as navigation aid. A class view abstracts from actual user interface details. In cases where the description of a class view does not sufficiently characterize the situation, e.g. if domain specific renderings are required, graphical drafts may additionally be attached.

Example: The left side of Fig. 4 shows the use case Written Exam Registration and a part of its activity graph. The right side depicts three scenes, one of them containing a class view, and a «*dependsOn*» relationship.

3.2. UIA dynamics: User Operations

To model the dynamic dialogue behaviour of the user interface, i.e. the navigation between scenes, the selection or manipulation of presented (displayed) domain objects, we adapt the notion of a user operation from FLUID to the UML-based SCORES+ modelling environment.

In SCORES+, a scene provides two different kinds of *user operations*:

- A user can (explicitly) navigate from one scene to another. The according user operation is stereotyped as «*navigation*».
- A «*manipulation*» denotes selection or editing of instances of domain classes which are presented in some class view of the scene. If a «*manipulation*» has triggered the root operation of the interaction related to the scene, an implicit navigation may happen. Such an implicit navigation is triggered by the system and not by the user.

A «*navigation*» can be marked as *creation* or *deletion*. In case of a *creation*, the target scene must be marked as *multiple*. Here the user's explicit change of focus creates a new instance of the target scene together with a navigation to that instance.

Stereotype	Description	Visualization
«*scene*»	A class stereotyped «*scene*» represents a main building block of the user interface.	Scene
«*class view*»	A class stereotyped «*class view*» is an abstract presentation of instances and relations of a domain class.	Class view
«*manipulation*»	An operation stereotyped «*manipulation*» denotes selection or editing of presented instances of domain classes.	Manipulation

Table 1. UIA elements as stereotypes

A «*navigation*» marked as *deletion* deletes the source scene when the user changes the focus to the target scene. Note that *creation* concerns the target scene of a «*navigation*», whereas *deletion* relates to the source scene. A scene in a set of «*mutually exclusive*» associated scenes is implicitly deleted, when the user changes the focus to another scene in the set. A scene which is part of a «*dependsOn*» association is also implicitly deleted, if the aggregating scene is deleted.

"Semanticless" user operations like window moving and re-sizing or scrolling are not considered. Like the other

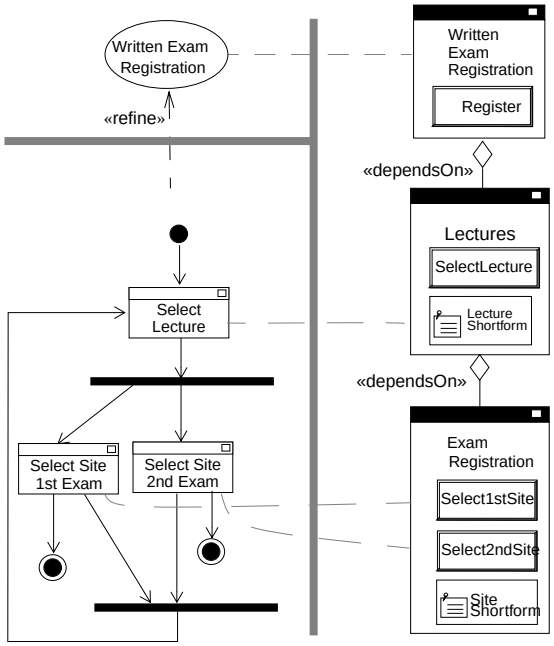

Fig. 5. Use case Written Exam Registration with activity graph and UIA elements (actors suppressed)

UIA elements, user operations abstract from concrete user interface details. For example, the particular interaction technique invoking a user operation, e.g. menu item selection or drag and drop, is not specified. Likewise, low level interactions like mouse button presses or basic text editing operations are not considered.

The UIA elements can be modelled as elements of a class diagram or as object flow states [29] in an activity graph. A «*navigation*» is simply visualized by a derived association between the according scenes. Tab. 1 comprises the visualizations of UIA elements together with their short descriptions.

Example: The right part of Fig. 5 depicts the UIA elements for the use case Written Exam Registration and for the according activity graph. The upper right scene is the start scene of the use case. Via a «*dependsOn*» relationship it aggregates the (sub-)scene Lectures which provides the «*manipulation*» SelectLecture(). The scene Lectures also contains the class view LectureShortform which presents a shorthand form of all lectures enrolled by the student. The scene Exam Registration contains the class view SiteShortform which represents short form information of all offered examination sites including room information. The scene Exam Registration is attached to both of the two interactions Select Site 1st Exam and Select Site 2nd Exam in the activity graph. The two «*manipulation*» user operations Select1stSite() and Select2ndSite() trigger the according root operations.

4 Prototyping

The coupling of SCORES elements and UIA elements (SCORES+ elements for short) aims at an integrated specification of functional and user interface requirements. For a semi-automatic generation of prototypes, however, the expressive power of SCORES+ elements is still not sufficient. To this end, we introduce the additional concept of a domain object variable forming the basis for the construction of meaningful prototypes.

4.1 Domain Object Variables

Class scopes and scenes of use cases or interactions deal with domain classes and not with concrete instances of that classes. The expressive power of such an approach is too weak to specify the semantics on the level of preciseness needed for a (semi-automatic) construction of prototypes. Here, the specification must come down to the object level. For example, it must be possible to express that objects like “the currently edited registration” or “the currently selected site” can manipulated, and corresponding application functionalities can be invoked.

To this end, we introduce domain object variables which are associated with class views. A *domain object variable* (DOV) is a UIA class that references a single domain object (stereotype «*singleDOV*») or a collection of domain objects (stereotype «*collectionDOV*») presented in the class view associated with the domain object variable. While a domain object variable references one or more instances of exactly one domain class (and its subclasses), a domain class may be referenced by more than one domain object variable. Tab. 2 describes the two kinds of DOVs.

Stereotype	Description	Visualization
«*singleDOV*»	Class stereotyped «*singleDOV*» references one instance of a domain class	Single DomainObject Variable
«*collectionDOV*»	Class stereotyped «*collectionDOV*» references several instances of a domain class	Collection DomainObject Variable

Table 2. Proposed stereotypes for domain object variables

A domain object variable can be associated with class views of more than one scene. This models the situation where the same domain object(s) is (are) presented in class views of different scenes. (To handle such situations was the main reason to introduce domain object variables.) For a class view of a multiple scene, the domain object variable can be specified as *local*. Then to every new instance of the scene, a new “instance“ of the domain object variable is attached to the new class view.

For a more precise specification of a «*manipulation*» user

operation, DOVs can be used as parameter types. According to the UML, each (DOV) parameter of a «*manipulation*» can be characterized as in, out, or inout, denoting read, creation, and update of the DOV. Additionally, DOVs can be associated with each other using the «*content related*» association (aggregation), denoting that domain objects referenced by the part DOV are also referenced by the aggregate DOV. For example, the (DOV) parameters of most selection operations are content related.

Example: Fig. 6 shows the scene Written Exam Registration and its dependent scene Lectures from Fig. 5, the latter complemented by the DOVs All Lectures and Selected Lecture. The «*collectionDOV*» All Lectures contains references to all lectures available to the student. The «*singleDOV*» Selected Lecture references the lecture which is currently selected by the student. Selected Lecture is «*content related*» to All Lectures because the selected lecture has to be among all lectures. With the «*manipulation*» SelectLecture() the user selects one of the lectures referenced by All Lectures and the selected lecture is then referenced by Selected Lecture. Accordingly, the parameter types of the «*manipulation*» SelectLecture() are All Lectures and Selected Lecture.

4.2. Prototype Generation

The generation of prototypes from the SCORES+ SRS and the DOVs takes place in two steps. The first step automatically transforms the SRS and the DOVs into the more detailed user interface design (UID) model. The UID model specifies, for example, abstract widget types and how a navigation is invoked (e.g. via menu choice or push button). The second step automatically generates an executable prototype for a target platform from the UID model.

The UID model specifies static structure and dialog behaviour of the user interface tier in the setting of a multi-tier architecture in a platform-independent style. It provides three fundamental modelling elements. A *UI-Spec* is a hierarchical composition of abstract widgets like canvas, button, text field, label, and so on. Each UI-Spec is attached to a *UI-Client* which specifies the dialog behaviour of a functionally cohesive part (normally a use case) of the user interface. UI-Clients are often hierarchically arranged, so we can talk about a "UI-Superclient" and a "UI-Subclient". *Objectholder* accommodate domain objects from the business logic tier of the application. To make the domain objects accessible from the UI-Clients, objectholders are associated with them. An objectholder associated with a "UI-Superclient" is also accessible from each of its "UI-Subclients".

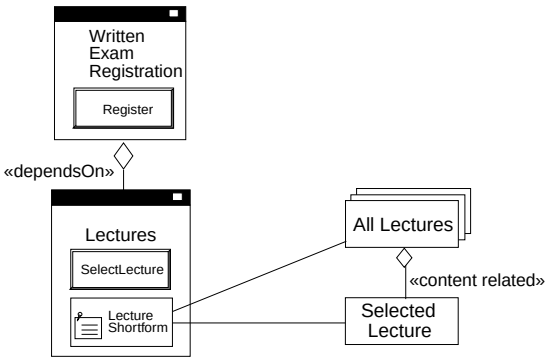

Fig. 6. UIA scene class with domain object variables

The first step of the prototype generation process starts with assembling a UI-Spec for each scene. To this end the class views of the scene become (sets of) abstract widgets. The widget types like e.g. TextField or NumericField are derived from the attribute types of the domain objects presented in the class views (e.g. String or Integer) and the types of the DOVs involved («*singleDOV*» and «*collectionDOV*»). Each user operation is also mapped onto an abstract widget, e.g. menu or push button. Note that the execution of a user operation belongs to the responsibilities of the UI-Client the UI-Spec is attached to.

For each start scene of a use case, a UI-Client is generated. The UI-Specs of the start scene and the UI-Specs of all scenes related to the activity graph of the use case are attached to the UI-Client. If a scene *T* «*dependsOn*» a scene *S*, the UI-Spec for *T* is embedded into the UI-Spec for *S*, except the UI-Specs for *S* and *T* are attached to different UI-Clients.

For the user operations of a scene, pseudo code is generated and collected in a command table contained in the client. In case of a «*navigation*» the pseudo code generation process takes also the «*dependsOn*» and «*mutually exclusive*» associations into account. If two scenes belonging to different use cases (resp. to their activity graphs) are «*dependsOn*» associated, the corresponding UI-Clients are hierarchically related accordingly.

Finally, for each DOV an objectholder is generated. The UI-Clients the objectholder is to be associated with are determined as follows. Let *S* be the set of all scenes containing class views the DOV is associated with. Then the objectholder will be associated with all UI-Clients to which UI-Specs of scenes of *S* are attached.

Usually, the automatically generated UID model will be "polished" manually to refine layout and navigation.

The second step of the generation process starts with choosing a target implementation platform. So far, our generation tool supports VisualWorks MVC [35] and Java Swing. A support for HTML-based platforms is currently under work. The generator substitutes (the abstract widgets of) the UI-Specs by the concrete widgets of the target platform (if necessary, an abstract widget is mapped onto a couple of concrete widgets). The tool also generates executable code from (the pseudo code of) the command tables of the UI-Clients. The objectholders are transformed

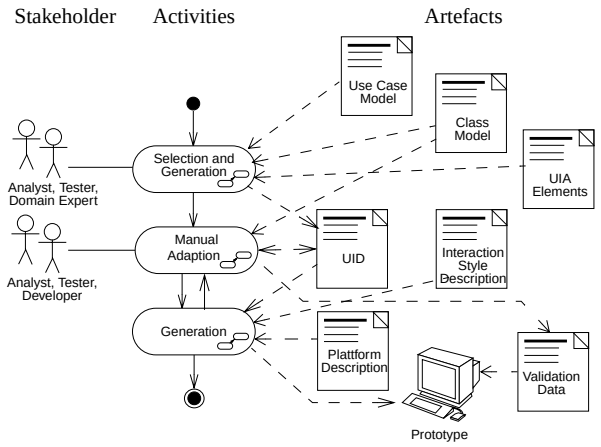

Fig. 7. Prototype construction process

into executable code, too.

Finally, domain object are generated from the domain class model and fed into the prototype [31].

For each supported platform we have developed a specialized framework that simplifies the code generation process. Such a framework comprises objectholder classes, (abstract) UI-Client classes, and some complex widgets not provided by the platform. The generated prototype is executable on a real device like a desktop computer running SUN CDE or MS Windows, a PDA supporting HDML [36], or a WAP/WML [37] compatible handy (simulator).

Fig. 7 shows the three activities of the semi-automatic generation of prototypes, the artifacts and the stakeholders involved. An important aspect of the generation process is the traceability from each element of the prototype via the UID model back to the SRS.

Example: The window in the back of Fig. 8 shows the UI-Spec editor depicting the hierarchical composition of the UI-Spec WrittenExamRegistrationSpec. Because of the «*dependsOn*» association between the start scene WrittenExamRegistration and the scene Lectures, the UI-spec of the latter is embedded into the UI-Spec WrittenExamRegistrationSpec. The abstract widgets Lecture (label) and Lecture list (combo box) are derived from the *«manipulation»* SelectLecture(), the two DOVs All lecture and Selected lecture, and from the class view Lecture shortform.

The window on top shows a screen shot of the UID model editor. The displayed diagram represents the part of the WEBADMIN UID model which belongs to the Written Exam Registration use case. The UID model has automatically been generated from the SCORES+ model shown in Fig. 5 and Fig. 6. The UI-Client WrittenExamRegistrationClient has been generated from the start scene Written Exam Registration. The ExamRegistrationSpec has been assembled from the scene Exam Registration and is a

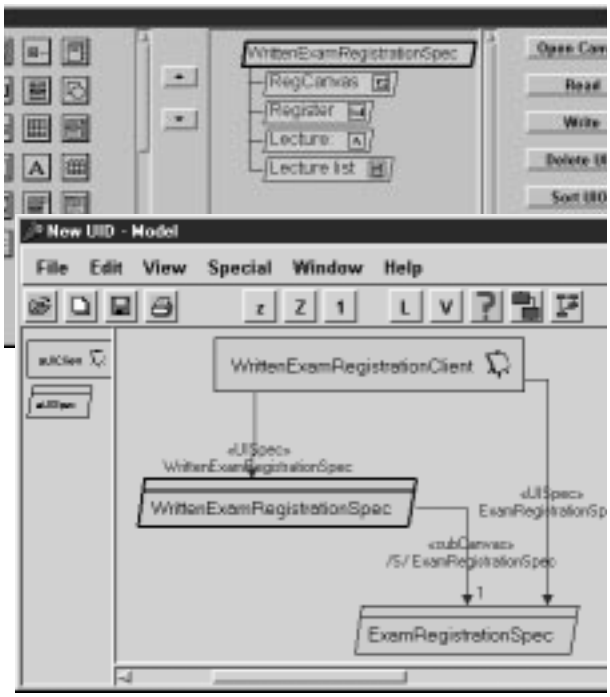

Fig. 8. UID editor with generated UID elements

subcanvas of the UI-Spec WrittenExamRegistrationSpec. In the prototype to be generated such a subcanvas will be presented within its containing canvas.

After having chosen the WIMP platform, our tool transforms the UID model into an executable prototype which is fully integrated into the VisualWorks® environment and can be manipulated with the VisualWorks® IDE [15]. The prototype is invoked by instantiating the main UI-client class in the VisualWorks® environment.

The left part of Fig. 9 depicts a screen shot of a window of the WIMP-style prototype. In the right part of Fig. 9, a screen shot of the prototype for a WAP/WML compatible handy simulator is shown. The handy prototype was built up by a hand crafted simulation of the generation process.

4.3. Validation and "Round-Trip" Prototyping

Before a validation session starts, analysts, testers, and domain experts select the part of the SRS to be validated and a particular target platform. Based on these decisions our prototyping tool generates the executable prototype. The validation session comprises a series of walks through business scenarios conducted with the help of the prototype. The validation proceeds analogously to the document based process described in [21]. During the validation, the prototype can be modified by an assisting developer according to the actual findings. Typically, further navigation relations between scenes or attributes to class views are added. The changes can be accomplished in the prototype, the UID model, or the SRS. In the two latter

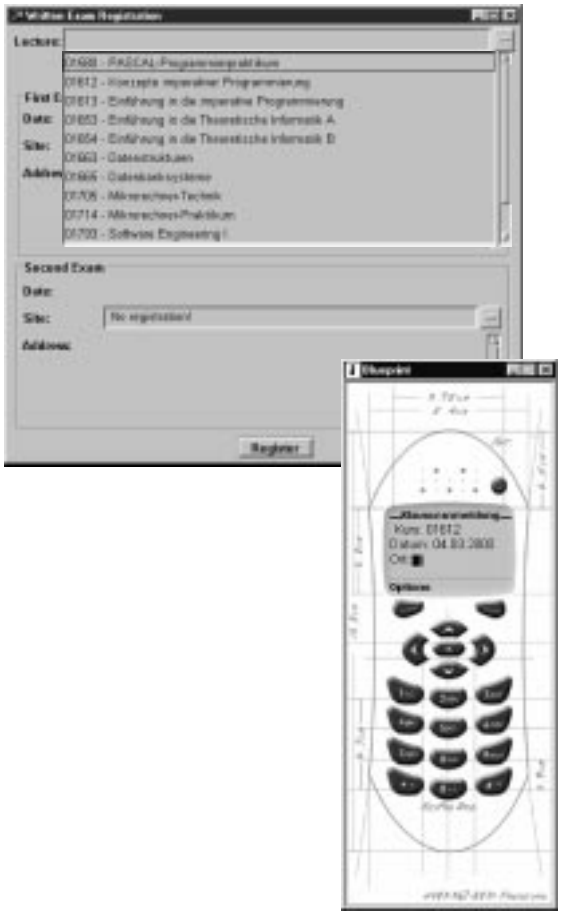

Fig. 9. Screen shots of a windows- and a handy prototype

cases, an incremental re-generation of the prototype takes place.

If the prototype is modified, the prototyping tool automatically feeds the modifications back into the UID model. The UID model updates are than mapped back to the SCORES+ SRS. For example, a change of a dialogue box triggers a change of the corresponding class view and domain class, and a change of a window from modal to non-modal triggers a change of the according scene. A modification of the navigation structure is mainly reflected by a change of the use case model. The round-trip prototyping brings the prototype and the SRS in sync and the prototype may be discarded when the validation process is finished. Fig. 10 shows the stakeholders, activities and artifacts of the round-trip prototyping process.

Example: Suppose that the validation reveals the following missing requirements: *Foreign students are allowed to write the examinations in an embassy; handicapped students are allowed to stay at home under the supervision of an official person*. To incorporate (and validate) these requirements, the prototype is manually extended with the VisualWorks canvas tool (Fig. 11). The canvas under work is the VisualWorks presentation of the UI-Spec WrittenExamRegistrationSpec. An abstract navigation widget Special Case is added together with a new UI-Spec SpecialCaseRegistrationSpec.

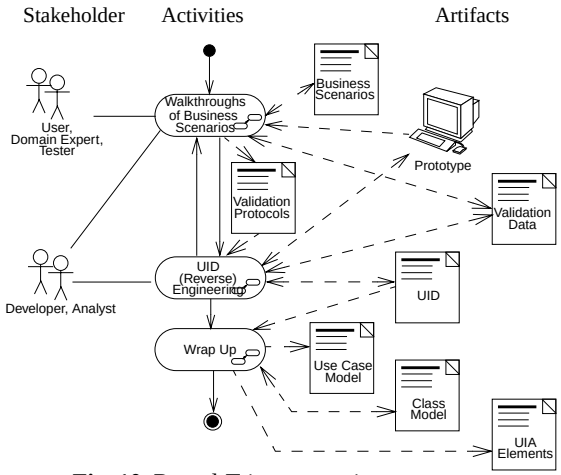

Fig. 10. Round-Trip prototyping process

Our prototyping tool feeds these modifications back into the UID model as follows. First, the modified and new widgets and UI classes of the executable prototype are transformed back into UI-Specs of the UID model. In case of the new UI-Spec SpecialCaseRegistrationSpec, the rule-based consistency checker detects that it is not connected to some UI-Client. A new UI-Client is then automatically created, named analogously, and the UI-Spec is attached to this UI-Client.

Fig. 11. Manipulating the WIMP user interface prototype with the VisualWorks CanvasTool

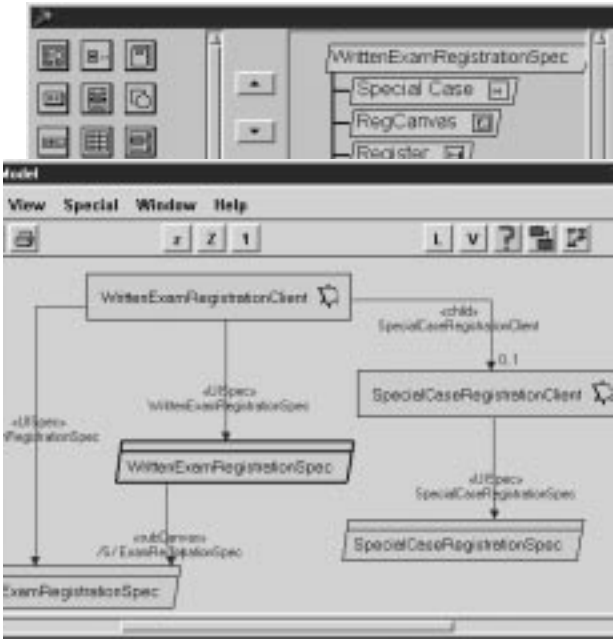

Fig. 12. UID editor with the updated UID model

In a second step, the modifications of the UID model are automatically mapped back to the SCORES+ SRS. The mapping includes the generation of a new scene SpecialCaseRegistration, which is target of a new *«navigation»* starting from the scene Lectures. Fig. 13 depicts the updated UIA elements.

A consistency check of the SCORES+ SRS now reveals a scene that is not connected to a use case or an action (in an activity graph). In our example, a new use case Special Case Registration is now manually added which *«extend»* the use case Written Exam Registration. The *«extend»* association implies the manual creation of the «*macro action*» Special Case Student in the according activity diagram. Fig. 14 shows the updated use case diagram and the modified activity graph of use case Written Exam Registration.

5 Experiences

We have applied the SCORES+ specification and validation techniques to a couple of projects ranging from small database applications over CASE tool components up to some main components of a "virtual university". This section mainly summarizes our experiences with the prototype-based validation of the SRS for WEBADMIN.

The SRS for WEBADMIN comprises 4 actors, 37 main use cases, 24 domain classes, and 28 main scenes. The presentation prototype for a window-based user interface architecture includes 13 windows with full navigation capabilities and some added application functionality like object creation and manipulation. Some simple WAP/ WML-based prototypes were built, too. The effort for the construction of the prototypes and the validation data was less than one person-week.

The prototypes were used and adapted in several validation sessions conducted for particular user groups. Most of the participating stakeholders were experts of the problem domain and used to window-based applications, and hence

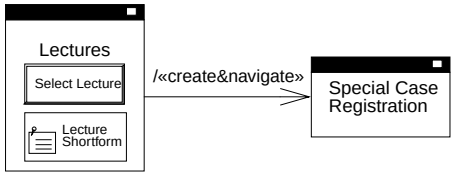

Fig. 13. Updated UIA elements

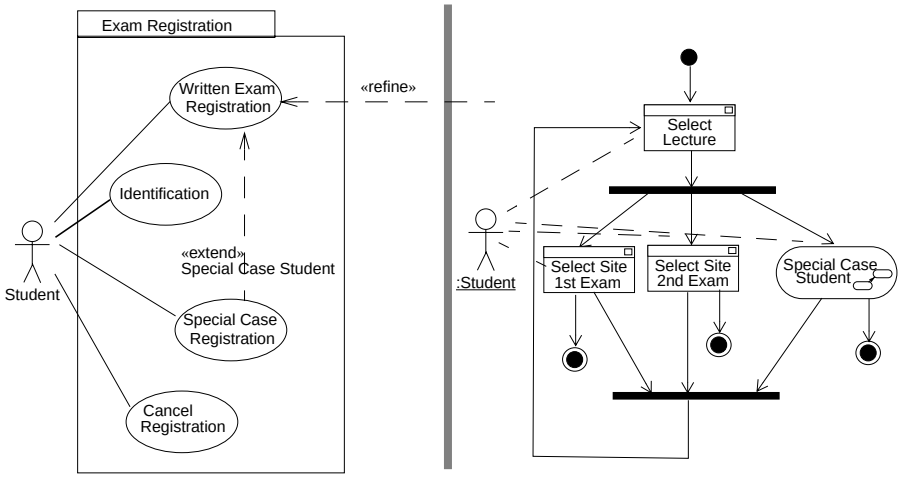

Fig. 14. Updated use case diagram and activity graph for use case Written Exam Registration

they did not encounter problems to identify the main tasks in the prototype. All stakeholders felt much more inclined in the prototype based validation process compared to a rather document centred one. In accordance to a study conducted by Cerpa and Verner [9], the analysts and especially the developers felt much more “safe“ during all development stages compared to former projects without prototype based validations.

Due to the prototype-based validation, we found omissions in the SRS which would presumably not have been found when using the original SCORES validation approach. For example, the initial SRS defined that a supervisor of a written examination belongs to the staff of the FernUniversität. This is in contrast to reality where supervisors of written examinations which, for example, take place in foreign countries, can be staff members of German embassies. During a validation session, some user asked for a “free form entry“ for supervisors which was then added to the prototype. It was easy to feed the new requirement via the UID model back into the SRS. It was materialized as a new interaction of the use case Specify Supervisor and a new domain class ForeignSupervisor. This class and the original domain class UniversityStaff became subclasses of a new class Staff.

In the initial SRS the actor Coordinator modelled the person that coordinates the rooms, supervisors, and the colours of the cover sheets of all written examinations to be conducted at a particular date. When we presented the prototype to some users from a particular faculty, they immediately asked, why the coordinator could not select the examinations for a particular semester. The reason was that in their faculty the coordinator is responsible to schedule all examinations of a semester, while in all other faculties the scheduling belongs to the responsibility of a member of the examination commission. By the way, even the inspections of the data dictionary had not detected this misconception.

Our experiences with prototypes for interaction techniques like WAP/WML browsers can be sketched as follows. In the requirements elicitation activities, stakeholders often came up with a bunch of needs concerning WAP/WML-based services. Fortunately, walking through some use cases with a WAP/WML browser prototype (as shown in the lower part of Fig. 9) helped a lot in separating real needed services from wishful thinking. We recommend an early prototyping of some of the WAP/WML-based services, which should be accomplished before negotiating the interaction styles for the whole bunch of use cases.

For the rest of the section we report on some pitfalls of the prototype-based validation of SRS. On the one hand, even for highly sophisticated window environments, only “rough“ prototypes should be constructed. This helps the users to focus on the functional requirements and prevents them from an early discussion about detailed layout and usability features. It also reduces the risks that users invent unrealistic expectations. On the other hand, for simple interaction styles like e.g. WAP/WML, every prototype bears the danger of too early in-depth discussions about layout and usability, because at first glance the prototype appears to the user as the interface of the final application.

We share the common experience that users, when playing around with a prototype, usually demand for more and more functionality [14]. This is often due to the fact that the tangibility and modifiability of the prototype let them easily imagine increased functionality without considering the additional costs. In general “users do seem to always ask for more“ [30]. The most important functionalities may be added to the prototype and validated at once, while others are either discarded or first incorporated in the SRS and validated later. In any case, each new functionality has to be negotiated.

6 Related Work

As mentioned in the introduction, Ravid and Berry aim at reconciling the difference between the prototype and other SRS documents [30]. Based on a case study, they propose a generic approach to taylor the process of prototyping. The main idea is to identify the kinds of requirements information the prototype should, respectively should not address before it is constructed. This information is called “intent-information“ and should help answering the question “When the prototype and the SRS disagree, which do one believe?“.

Schneider proposes a strategy to extract crucial pieces of knowledge from a prototype and from its developer [33]. The strategy is based on monitoring the explanations given by the developer, analysing their structure, and feeding results back to support and to focus the explanations. During this process, the prototype turns into the centrepiece of a hyperstructured information base which can be used to convey concepts, implementation tricks and experiences.

Both approaches are different from our attempt and the problem tackled by them seems to disappear or at least to heavily be reduced by our “round-trip“ facility synchronizing the prototype and the SRS.

Kovacievic [22] and Nunes [27] present extensions of the UML for task and user interface modelling. Conallen proposes UML extensions and a process to model and develop HTML-based web applications [10]. These approaches neither focus on prototyping nor aim at an integration of the functional and the user interface requirements.

Elkoutbi et al. suggest a process that generates a user interface prototype from scenarios via an intermediate state-based specification of the system [12]. Scenarios of use cases are modelled as UML collaboration diagrams enriched with user interface information. For example, messages are labelled with an optional “widget mark“

indicating the widget-type the resulting user interface should provide. A drawback of their approach is the tedious manual task of specifying all scenarios on the level of widget interactions. The requirements engineer is forced to have a concrete user interface in mind from the beginning. Another disadvantage is the missing support of user interaction techniques other than menus and windows. The approach also lacks the mapping of validation results from the (modified) prototype back to the requirements specification.

Many software tools for the generation of user interfaces are known [26]. Some of them, e.g. Lauesen's EFDD approach [23] or Balzert's Janus system [3], use only the structural (data) model. Others, e.g. Johnson's ADEPT [18] or the approach of Bomsdorf and Szwillus [5], are based only on a functional (task) model. A few approaches, e.g. Forbrig's and Schlungbaum's TADEUS [13] or the Genova system by Arisholm et al. [1], are based on functional and structural models. All these approaches focus on the generation of full fledged user interfaces as a starting point for a complete application generation. They are not concerned with the validation and none of them allows for a round-trip prototyping.

Formal specification based prototyping approaches like e.g. the CAPS project by Luqi [24] mainly focus on the (semi-) automatic generation of functional prototypes. Presentation prototypes are either neglected or must be created manually.

7 Conclusion

We have proposed SCORES+, a requirements specification method that covers functional and user interface requirements in an integrated and UML-compliant way. A SCORES+ SRS also forms the basis for the semi-automatic construction of prototypes running on different target platforms. In addition, the generation technique allows for "round-trip" prototyping.

At the moment, we are applying our approach to several problem domains for collecting more experiences and for further validation of the method. We are also developing a framework that conveniently supports web-based environments as implementation platform for prototypes. Main future work concerns the seamless transition of SCORES+ SRS into a software architecture and the generation of test cases for user interface and acceptance tests.

Acknowledgments

The authors like to acknowledge the comments of the anonymous referees which led to a considerable improvement of the paper. We also thank Johannes Akinlaja, who implemented the SCORES+ editor on top of the Together/J® [34] UML modelling tool.

References

1. E. Arisholm, H. Benestad, J. Skandsen, H. Fredhall: Incorporating Rapid User Interface Prototyping in Object-Oriented Analysis and Design with Genova. Proc. 8. Nordic Workshop on Programming Environment Research, Ronneby, Sweden, August 1998
2. D. Bäumer, W. Bischofberger, H. Lichter, H. Züllighoven: User Interface Prototyping — Concepts, Tools, and Experience. Proc. ICSE-18, Berlin, Germany, March 25 - 29 1996, pp. 532-541
3. H. Balzert et al.. The Janus Application Development Environment: Generating More than the User Interface. In: J. Vanderdonckt (ed.): Computer-Aided Design of User Interfaces, Namur University Press, Namur, 1996, pp. 183-205
4. Boehm B., Verifying and Validating Software Requirements and Design Specifications. IEEE Software, Jan. 1984, pp. 75-88
5. B. Bomsdorf, G. Szwillus: Coherent Modelling & Prototyping to Support Task-Based User Interface Design. CHI'98 Workshop "From Task to Dialogue: Task-Based User Interface Design", Los Angeles, California, 1998
6. G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language Users Guide. Addison-Wesley/acm Press, Reading, MA, 1999
7. J. Brunsmann, A. Homrighausen, H.-W. Six, J. Voss. Assignments in a Virtual University — The WebAssign System. Proc. 19th World Conf. on Open Learning and Distance Education, Vienna, Austria, 1999
8. J. M. Carroll (Ed.): *Scenario-Based Design — Envisioning Work and Technology in System Development*. J. Wiley & Sons, New York, 1995
9. N. Cerpa, J. Verner: Prototyping: some new Results. Information and Software Technology, Vol 38, Elsevire, 1996, pp. 743-755
10. Jim Conallen: Building web applications with UML. Addison-Wesley object technologiey series, 2nd printing, February 2000
11. A. Davis: Software Prototyping. in: M. Yovits, M. Zelkowitz (Eds.): Advances in Computers, Vol. 40, Academic Press, San Diego, 1995, pp. 39-63
12. M. Elkoutbi, I. Khriss, R.K. Keller: Generating User Interface Prototypes from Scenarios. Proc. RE 99, IEEE Computer Society Press, 1999, pp. 150-158
13. P. Forbrig, C. Stary et al.: From Task to Dialog: How Many and What Kind of Models do Developers Need? Proc. CHI'98 Workshops, April 1998
14. V. S. Gordon, J.M. Bieman: Rapid Prototyping: Lessons Learned. IEEE Software, Vol. 15, Nr. 1, Jan. 1995, pp. 85-95
15. A. Homrighausen, J. Voss: Tool support for the model-based development of interactive applications - The FLUID approach. Proc. 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems, Granada, 1997

16. IEEE *Guide to Software Requirements Specifications* IEEE Standard 830, IEEE, New York, 1993
17. I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process. Addison-Wesley/acm Press, Reading, MA, 1999
18. P. Johnson, H. Johnson, S. Wilson: Rapid Prototyping of User Interfaces Driven by Task Models. Chapter 9 in [8].
19. G. Kösters, B.-U. Pagel, T. de Ridder, M. Winter: Animated Requirements Walkthroughs based on Business Scenarios. Proc. 5th euroSTAR, Edinburgh, 1997
20. G. Kösters, H.-W. Six, J. Voss: Combined Analysis of User Interface and Domain Requirements. Proc. 2nd IEEE Int. Conf. on Requirements Engineering, Colorado Springs, 1996
21. G. Kösters, H.-W. Six, M. Winter: Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications. Requirements Engineering, Vol. 6, Nr. 1, Springer, London, January 2001, pp. 3-17
22. Srdjan Kovacevic: UML and User Interface Modeling. In: Jean Bezivin, Pierre-Alain Muller (eds.): <<UML>>'98 The Unified Modeling Language, Lecture Notes in Computer Science 1618, 1998, pp.253-266
23. S. Lauesen, M. B. Harning, C. Grønning: Screen Design for Task Efficiency and System Understanding. In: S. Howard and Y.K. Leung (eds.): OZCHI 94 Proceedings, 1994, pp. 271-276
24. Luqi, et al.: CAPS as Requirements Engineering Tool. Proc. RE and Analysis Workshop, SEI, CMU, Pittsburgh, 1991 (http://wwwcaps.cs.nps.navy.mil)
25. N.A.M. Maiden, S. Minocha , K. Manning, M. Ryan: CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements. Proc. ICRE98, IEEE Press, 1998
26. B. Myers, S. E. Hudson, and R. Pausch: Past, Present, and Future of User Interface Software Tools. ACM Transactions on Computer-Human Interaction, Vol. 7, No. 1, March 2000, pp. 3-28
27. Nuno Jardim Nunes, Joao Falcao e Cunha: Towards a UML Profile for Interaction Design: The Wisdom Approach. Proc. <<UML>>2000 The Unified Modeling Language, York, UK, Oct. 2000
28. B. Nuseibeh, S. Easterbrook: Requirements Engineering: A Roadmap. ICSE-18, The Future of Software Engineering, Limerick, Ireland, 2000, pp. 35-46
29. Object Managment Group: Unified Modeling Language Specification. Version 1.3, OMG, June 1999
30. A. Ravid, D. M. Berry: A Method for Extracting and Stating Software Requirements that a User Interface Prototype Contains. Requirements Eng 5 (2000) 4, pp. 225-241
31. J. Rogotzki: Generating Domain-Data from GeoOOA-Models. Diploma thesis, Dept. of CS, FernUniversität Hagen, 1997 (in German)
32. M. B. Rosson, J. M. Carroll: Narrowing the Specification-Implementation Gap in Scenario Based Design. Chapter 10 in [8].
33. K. Schneider: Prototypes as assets, not toys: why and how to extract knowledge from prototypes. Proc. ICSE 18, 1996, pp. 522 - 531
34. TogetherJ ControlCenter V.5.02, TogetherSoft Corporation, 2001
35. VisualWorks Release 5i.1, Cincom Systems Inc., 1999
36. W3C: Handheld Device Markup Language Specification. May 1997 http://www.w3.org/pub/WWW/TR/NOTE-Submission-HDML-spec.html
37. WAP-Forum: Wireless Markup Language Specification. WAP-191, June 2000, http://www.wapforum.org
38. Weidenhaupt K, Pohl K, Jarke M, Haumer P. Scenarios in System Development: Current Practice. IEEE Software, March/April 1998, pp 34-45