Development of Graphical User Interfaces based on User Interface Patterns

Stefan Wendler, Danny Ammon, Teodora Kikova, Ilka Philippow
Software Systems / Process Informatics Department
Ilmenau University of Technology
Ilmenau, Germany
{stefan.wendler, danny.ammon, teodora.kikova, ilka.philippow}@tu-ilmenau.de

***Abstract* — This paper addresses the research concerning possibilities for reducing the effort of adapting graphical user interfaces to requirements of individual customers. User interface patterns are promising artifacts for improvements in this regard. The details of graphical user interface transformations from user interface patterns into executable interface code are considered. We describe how reuse and automation within user interface transformation steps can be established. For this purpose, formal descriptions of user interface patterns are necessary. Today, however, most user interface patterns exist only in a verbal or graphical form of description. We use XML-based user interface description languages like UIML and UsiXML for the specification of user interface patterns. We experimentally investigated and analyzed strengths and weaknesses of two transformation approaches which were built on different software patterns. As a result, we show that formal user interface patterns can be transformed into executable interfaces, and that they assist in raising effectiveness and efficiency of the development process of a GUI system. Finally, we developed suggestions on how to apply these positive effects of user interface patterns for the development of pattern-based graphical user interfaces.**

Keywords — graphical user interface; model driven software development; user interface patterns; UIML; UsiXML

I. INTRODUCTION

Interactive systems. Interactive systems demand for a fast and efficient development of their graphical user interface (GUI), as well as its adaption to changing requirements throughout the software life cycle. In this paper, e-shops serve as a representative of these interactive systems. Currently, they are a fundamental asset of modern e-commerce business models. In many cases, such systems are offered as standard software, which allows several customization options after installation. In this context, they are differentiated into the application kernel and a GUI system.

The application kernel software architecture relies on well-proven and, partially, self-developed software patterns. Thus, it offers a consistent structure with defined and differentiated types of system elements. This has a positive effect on the understanding of the modular functional structures as well as their modification options.

Limited customizability of GUIs. Contrary to the application kernel, the customization of the GUI is possible only with rather high efforts. An important reason is that software patterns do not cover all aspects needed for GUIs. These patterns have been commonly applied for GUIs [1][2] but in most cases they are limited to functional and control related aspects [3]. The visual and interactive components of the GUI are not supported by software patterns yet. Furthermore, the reuse of GUI components, e.g., layout, navigation structures, choice of user interface controls (*UI-Controls*) and type of interaction, is only sparsely supported by current methods and tools. For each project with its varying context, those potentially reusable entities have to be implemented and customized anew leading to high efforts.

Moreover, the functional range of standard software does not allow a comprehensive customization of its GUI system. The GUI requirements are very customer-specific. In this regard, the customers want to apply the functionality of the standard software in their individual work processes along with customized dialogs. However, due to the characteristics of standard software, only basic variants or standard GUIs can be offered. So far, combinations of components of the application architecture with a GUI are too versatile for a customizable standard product.

UIPs. We propose an approach to this problem through the deployment of User Interface Patterns (UIPs). These patterns offer well-proven solutions for GUI designs [4], which embody a high quality of usability [5]. So far, UIPs have not been considered as source code artifacts, in contrast to software patterns. Current UIPs and their compilations mostly reside on an informal level of description [6].

A. Objectives

In this paper we show that formal UIPs can assist in raising effectiveness and efficiency of the development process of a GUI system. For a start, we describe, from a theoretical point of view, how reuse and automation within GUI transformation steps can be established by the deployment of UIPs. On the basis of formal UIPs, we discuss the possibilities of transformations into executable GUIs. For this purpose, two different transformation approaches have been experimentally investigated. These approaches will be assessed facing two different GUI dialogs. As a result, we develop suggestions, how the positive effects of UIPs for the development of GUIs can be applied. Finally, influences resulting from the use of UIPs in the development process are discussed.

B. Structure of the Paper

In Section II, state of the art and related work are presented and assessed according to our objectives. The theoretical influences of UIPs on the development process

 ISBN: 978-1-61208-221-9

for GUIs are elaborated in Section III. Subsequently, Section IV presents our two approaches for the transformation of formal UIPs into source code. The findings of Sections III and IV are summarized in Section V. Finally, our conclusions and future research options are presented in Section VI.

II. RELATED WORK

A. GUI Development Process and Model Transformations

Abstract GUI development model. The specification and development of GUI systems remains a challenge. To discuss the activities and potentials of UIPs independently from specific software development processes and requirement models, we refer to a generic model concept. In reference [7], the common steps of a GUI development process are elaborated. To master the complexity that occurs when deriving GUI specifications from requirement models, Ludolph proposes four model layers and corresponding transformations built on each other. Three of them, being relevant in our context, are depicted in Figure 1.

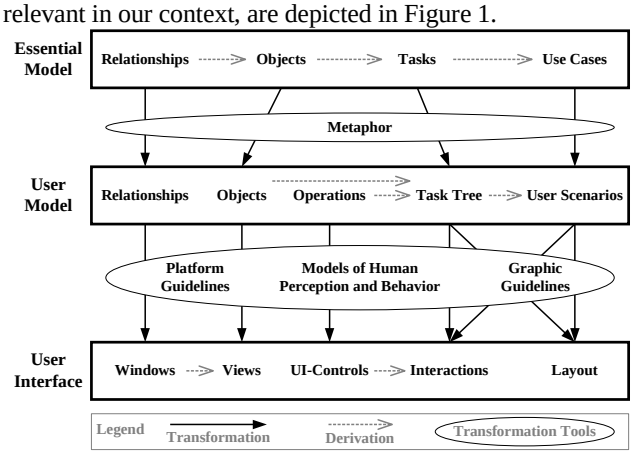

Figure 1. Model transformations in the GUI development process based on [7]

Essential model. By the e*ssential model*, all functional requirements and their structures are described. This information constitutes the core of the specification which is necessary for the development of the application kernel. Examples for respective artifacts are *use cases*, domain models and the specification of *tasks* or functional decompositions. These domain-specific requirements are abstracted from realization technology and thus from the GUI system [7]. Consequently, a GUI specification must be established to bridge the information gap between requirements and a GUI system.

User model. A first step in the direction of GUI specification is prepared by the *user model*. With this model the domain-specific information of the *essential model* is picked up and enhanced by so-called *metaphors*. They symbolize generic combinations of actions and suitable tools, which represent interactions with a GUI. Examples of *metaphors* would be indexes, catalogues, help wizards or table filters. The principal action performed by these examples is a search for objects, accompanied by the varying functionality embodied by the respective *metaphor*.

The *tasks* of the *essential model* have to be refined and structured in *task trees*. For each *task* of a certain refinement stage, *metaphors* are assigned, which will guide the GUI design for this part of the process. In the same manner, *use cases* can be supplemented with these new elements in their sequences to describe *user scenarios*.

User interface. This model is used for establishing the actual GUI specification. Through the three parts rough layout, interaction design and detailed design [7], the appearance and behavior of the GUI system are concretized. The aim is to set up a suitable mapping between the elements of the *user model* and *views*, *windows*, as well as *UI-Controls* of the *user interface*. For the *metaphors* chosen before, graphical representations are now to be developed. The *objects* to be displayed, their attributes and the relations between them are represented by *views*. Subsequently, the *views* are arranged in *windows* according to the activities of the *user scenarios*, or alternatively to the structure of the more detailed *task trees*. In these steps, there are often alternatives which are influenced by style guides or the used GUI library and especially by the provided *UI-Controls*. At the same time, generic interaction patterns are applied as transformation tools which also have an impact on the choice of *UI-Controls*.

Conclusion. Model transformations as stated by Ludolph show a detailed account of relevant model elements for the GUI specification. However, the occurring transformations are carried out manually. Besides that, no automation and only few options for reuse are mentioned.

B. UIP Definition and Types

Current research has been discussing patterns and especially User Interface Patterns (UIPs) for a longer period [8][9][6]. A UIP is defined as a universal and reusable solution for common interaction and visual structures of GUIs. UIPs are distinguished between two types:

Descriptive UIPs. Primarily, UIPs are provided by means of verbal and graphical descriptions. In this context, UIPs are commonly specified following a scheme similar to the one used for design patterns [10]. Reference [11] proposes a specialized language for UIPs and [6] shows its detailed sections. The verbal descriptions mainly serve for pure specification purposes and solely fulfill an informational function for the GUI developer. Being a guideline in this manner, they provide templates, points of variability and sketched examples for GUI elements. These UIPs named as descriptive UIPs [6] are informal. With their application, a developer receives aid when specifying a GUI, as he is able to express and hence operationalize usability requirements with UIPs. However, these informal patterns still have to be implemented manually.

UIP-Libraries. UIP libraries such as [12], [13] and [14] provide numerous examples for descriptive UIPs. Based on the presented categories, conceptions about possible UIP hierarchies and their collaborations can be imagined.

Formal UIPs. Rarely, generative UIPs [6] are presented. In contrast to descriptive UIPs, they feature a machine-

 ISBN: 978-1-61208-221-9

readable form and are regarded as formal UIPs accordingly. Frequently, the formal format constitutes of a graphic notation, e.g., UML [8]. The formal UIPs are of great importance since they can be used within development environments, especially for automated transformations to certain GUI-implementations.

C. Formalization of UIPs

In order to permit the processing of descriptive UIPs, they have to be converted to formal UIPs. Possible means for this step can be provided by formal languages applied for specifying GUIs. These languages, however, have been designed for the specification of certain GUIs and were not intended for a pattern-based approach. Until now, there is no specialized language available for formalizing UIPs.

UsiXML and UIML. In our prior work, an extensive investigation on formal GUI specification languages and their applicability for UIPs was conducted. Intentionally the XML-based languages UsiXML [15] and UIML [16] were developed for specifying a GUI independently from technology and platform specifics. However, such languages may be applicable for UIPs since they offer elements like templates (UIML) and abstract as well as concrete models (UsiXML). Moreover, both have been developed further for a long period of time. Thus, the languages have reached a high maturity level.

IDEALXML. For efficient development environments tools are necessary that facilitate formal specifications of UIPs with regard to language definitions and rules. A widespread tool concept for UsiXML is presented with IDEALXML [6]. By using the various models defined by UsiXML, many aspects of a GUI and additionally the applied domain model of the application kernel are included in the specification. As a result, a detailed and comprehensive XML specification for the GUI is created. Many aspects of the *user model* from [7] are already included. However, it is not mentioned how UIPs are being expressed in models such as the „abstract user interface model“ (AUIM) [6] as reusable patterns or an hierarchy of these and consequently transformed to the „concrete user interface model“ (CUIM) [6].Furthermore, it has to be questioned, how a formal specification on the basis of UsiXML can be used for processing by code generators or other tools of a development environment.

D. GUI-Generators

Besides the formal specification of GUIs system concepts and frameworks exist which are able to generate complete GUI applications based on a partly specification of the application kernel. As representatives Naked Objects [17] and JANUS [18] can be mentioned. Both rely on an object-oriented domain model which has to be a part of the application kernel. Based on the information provided by this model, standard dialogs are being generated with appropriate *UI-Controls* for the respective *tasks*. For instance, in order to generate an *object* editor for entities like product or customer, certain text fields, lists or date pickers are selected as *UI-Controls* which match the domain data types of the selected domain *object* for editing.

In contrast to IDEALXML, which enables the extensive modeling of the GUI, GUI-generators may generate executable GUI code but they lack such a broad informational basis. Therefore, GUI-generators possess two essential weaknesses:

Limited functionality. The information for generating the GUI is restricted to a domain model and previously determined dialog templates along with their *UI-Controls*. Hence, their applicability is limited to operations and relations of single domain *objects*. When multiple and differing domain *objects* do play a role in complex *user scenarios* [7], the generators can no longer provide suitable dialogs for the GUI application. Moreover, extensive interaction flows require hierarchical decisions, which have to be realized, e. g., by using wizard dialogs. In this situation, GUI generators cannot be applied as well. The connection between dialogs and superordinate interaction design still has to be implemented manually.

Uniform visuals. A further weakness is related to the visual GUI design. Each dialog created by generators is based on the same template for the GUI-design. Solely the contents which are derived from the application kernel are variable. Both *layout* and possible *interactions* are fixed in order to permit the automatic generation. The uniformity and its corresponding usability have been criticized for Naked Objects [19]. Assuming the best case, the information for GUI design is founded on established UIPs and possesses their accepted usability for certain *tasks*. Nevertheless, the generated dialogs look very similar and there is no option to select or change the UIPs incorporated in the GUI design.

III. Influence of UIPs on GUI-Transformations

A. GUI Customization of Standard Software

On the basis of the customization of GUIs for standard software and the model transformations described in Section II.A the theoretical influences of UIPs are now considered.

EShop standard software fulfils the functional requirements of a multitude of users at the same time. Therefore, these systems share a well-defined *essential model* that specifies their functional range and has many commonalities along existing installations. Standard software implements the *essential model* through different components of the application kernel as shown in Figure 2. Each installation consists of a configuration for the application kernel which includes many already available and little custom components in most cases. In this context, the *User Interface* acts as a compositional layer that combines *Core* and *Custom Services* together with suitable dialogs for the user.

Individual GUIs for eShops. Concerning eShops, the visual design of the GUI is of special relevance since the user interface is defined as a major product feature that differentiates the competitors on the market. Hence, the needs of customers and users are vitally important in order to provide them with the suitable individual dialogs. In this regard, the proportions of components related to the whole system are symbolized by their size in Figure 2. In comparison to the *Custom Components* of the application

 ISBN: 978-1-61208-221-9

kernel the *Custom Dialogs* represent the greater part of the *User Interface* and the customization accordingly. Along with the customization of the application kernel there is a high demand for an easy and vast adaptability of the GUI.

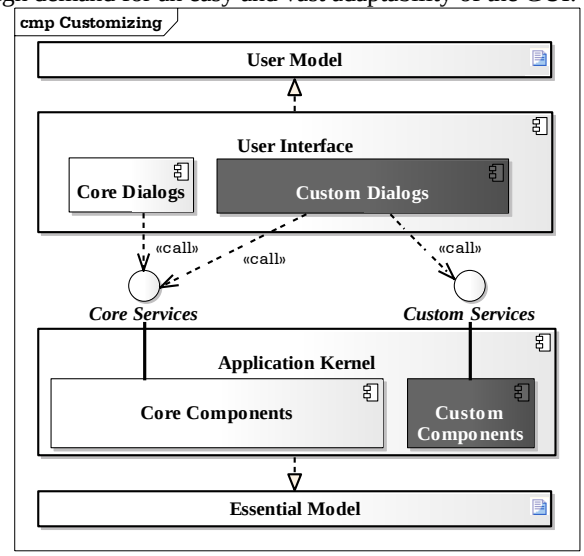

Figure 2. Components for the customization of standard software

Moreover, the customization of the GUI system is needed, as elements of the *essential model* tend to be very specific after extensive customization or maintenance processes. Thus, the standard *user model* as well as the *user interface* can no longer be used for the customized services. In this case, models have to be developed from scratch and after this, a suitable solution for the GUI has to be implemented.

Usability. The development of GUIs is caught in a field of tension between an efficient design and an easy but extensive customization. High budgets for the emerging efforts have to be planned. Additional efforts are needed for important non-functional requirements such as high usability and uniformity in interaction concepts and an eased learning curve during the customization process of GUIs. For realizing these requirements, extensive style guides and corresponding *user interface* models often need to be developed prior to the manual adaption of the GUI. These specifications will quickly lose their validity as soon as the GUI-framework and essential functions of the application kernel change.

B. Model Aspects of UIPs

With the aid of UIPs the time-consuming process of GUI development and customizing can be increased in efficiency. To prove this statement, the influences of UIPs on the common model transformations from Section II.A are examined in the next step. In Section III.C potentials for improvements are derived from these influences.

Metaphors and UIPs. *Metaphors* act as the sole transformation tool between *essential model* and *user model*. Since they lack visual appearances as well as concrete interactions, the mapping of *metaphors* to the elements of the *essential model* is very demanding. *Metaphors* will not be visualized by GUI sketches prior to the transformation of the *user model*.

Since UIPs are defined more extensively and concrete, they can be applied as a transformation tool instead of using *metaphors*. Descriptive UIPs feature a pattern-like description scheme that is provided in the catalogues in [12] and [13], for example. Thus, they offer much more information as well as assessments which can inspire the GUI specification. In addition, descriptive UIPs do already possess visual designs that may be exemplary, or in the worst-case, abstract.

With the *user model*, operations on *objects* have to be specified. The *metaphors* do not provide enough hints for this step. In contrast, UIPs are definitely clearer concerning these operations because they group *UI-Controls* according to their *tasks* and do operationalize them in this way. Interaction designs and appropriate visuals are presented along with UIPs. These aspects would have to be defined by oneself using only the *metaphor*.

When UIPs are used in place of *metaphors* for formalization, these new entities can be integrated in the tools for specifications. Concerning UsiXML, UIPs could describe the AUIM. *Task-Trees* are already present in UsiXML, so this concept of specification partly follows the modeling concepts in [7] and thus may be generically applicable.

User model and UIPs. With regard to the *user model*, the numerous modeling steps no longer need to be performed with the introduction of UIPs. Instead, it is sufficient to derive the *tasks* from the *use cases* within the *essential model* and allocate UIPs for these. Detailed *task-trees* no longer have to be created since UIPs already contain these operations within their interaction design. Interactions can already be specified in formal UIPs, and later this information can directly be used for parts of the presentation control of *views* or *windows*. As a result, an extensive *user scenario* also is obsolete, as it was originally needed for deriving the more detailed *task-tree*. Now it is sufficient to lay emphasis on expressing the features of UIPs and their connection to the *tasks* defined by the *essential model*. The *objects* are also represented within the UIPs in an abstract way. With the aid of placeholders for certain domain data types adaptable *views* for *object* data can already be prepared in formal UIPs. Finally, much of the afore-mentioned information of the *user model* now will be explicitly or implicitly provided by completely specified UIPs.

User interface and UIPs. UIPs provide the following information for the *user interface*: *Layout* and *interaction* of the GUI will be described by a composition of a hierarchy of UIPs that is settled on the level of *views* and *windows*. When creating the UIP-hierarchy, a prior categorization is helpful which features the distinction between *relationship*, *object* and *task* related UIPs. This eases the mapping to the corresponding model entities.

For *interactions*, the originally applied *Models of Human Perception and Behavior* from Figure 1 are no longer explicitly needed since they are implicitly incorporated in the interaction designs of the UIPs. In this context, suitable types

 ISBN: 978-1-61208-221-9

of *UI-Controls* are already determined by UIPs. Nevertheless, a complete and concrete GUI-design will not be provided by UIPs since the number and contents of *UI-Controls* depend on the context and have to be specified by the developer with parameters accordingly. In the same way *Platform* and *Graphic Guidelines* act as essential policies to adapt the UIPs to the available GUI-framework and its available *UI-Controls*.

Conclusion. We explained that UIPs might cover most parts of the *user model* as well as numerous aspects of the *user interface*. By using UIPs in the modeling process, these specification contents can be compiled based on the respective context without actually performing the two transformations from Figure 1 explicitly. Basically, the transformation to the target platform remains as depicted in Figure 3.

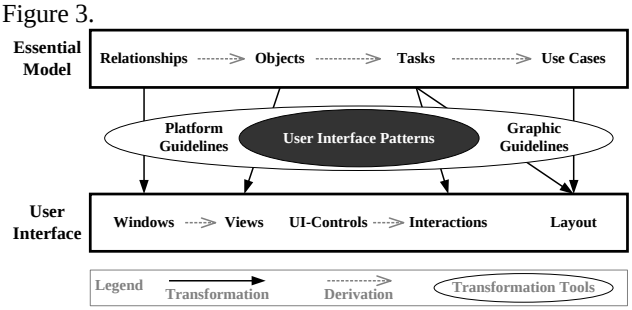

Figure 3. GUI transformations with the aid of UIPs and automation

C. Potentials of UIPs for Improvements

In this section, the potentials of UIPs related to the GUI development process are summarized from a theoretical perspective. The implications resulting from the application of UIPs in experimental transformations are presented in Section IV.

Reuse. By means of UIPs the transformational gap between *essential model* and *user interface* can be bridged more easily since reuse will be enhanced significantly. Thereby UIPs are not the starting point of model transformations; they rather serve as a medium for conducting needed information for the transformations. The information originally included in the *user model* and parts of the *user interface* are now extracted from the selection and composition of UIPs.

Layout and *interaction* of *windows* as well as the interaction paradigm of many parts of the GUI can be determined by a single UIP configuration on a high level in hierarchy. This superordinate GUI design can be inherited by a number of single dialogs without the need for deciding about these aspects for each dialog in particular.

Many interaction designs can be derived from initial thoughts about GUI design for the most important *use cases* and their corresponding *tasks*. When a first UIP configuration has been created, the realization of the *Graphic* and *Platform Guidelines* therein can be adopted for other UIP-applications since the target platform is the same for each dialog of a system. Especially when *user scenarios* overlap, meaning they partly use the same *views* or *windows* as well as *object* data, UIPs enable a high grade of reuse. UIP assignments, already established for other *tasks*, can be reused with the appropriate changes. eShops tend to use many application components together although they offer them by different dialogs as illustrated in Figure 2 UIPs can contribute to a higher level of reuse in this context. Depending on the possible mapping between application kernel components and UIP-hierarchy, new dialogs can be formed by combining the views of certain services which are determined by their assigned UIPs.

Reuse and usability. Besides reuse, UIPs assure multiple non-functional requirements. As proven solutions for GUI designs their essential function is to enable a high usability by the application of best-practices. In this context, they facilitate the adherence of style guides by means of their hierarchical composition.

Technically independent essential model. It is a common goal to keep elements of the *essential model* free or abstract from technical issues. Following this way, the *essential model* has no reference to the GUI specification. Therefore, it is not subject to changes related to new requirements which the user may incorporate for the GUI during the lifecycle of the system. User preferences often tend to change in terms of the visuals and interactions of the GUI. Concerning *use cases*, this rule is elaborated in [20] and [21]. Technical aspects and in particular the GUI specification are addressed in separate models such as *user model* and *user interface* according to [7]. After changes, these models have to be kept consistent what results in high efforts. For instance, a new or modified step within a *use case* scenario has to be considered in the corresponding *user scenario*, too.

By assigning UIPs to elements of the *essential model*, explicit *user models* and the prior checking of consistency between these models both become obsolete. Instead, *user models* will be created dynamically as well as implicitly by an actual configuration of UIPs and *essential model* mapping. A technical transformation to the source code of the GUI that relies on the concrete appearances of the UIPs remains as shown in Figure 3. By modeling assignments between UIP and *task* or between UIP and *object*, the number of *UI-Controls*, the hierarchy and *layout* of UIPs, sufficient structured information on the GUI system is provided. Subsequently, a generator will be able to compile the GUI suited for the chosen target platform. These theoretical influences enable an increased independence from the technical infrastructure since the generator can be supplied with an appropriate configuration to instantiate the UIPs compatible to the target platform and its specifics.

Modular structuring of windows and views. Common to software patterns, UIPs reside on different model hierarchies. Dialog navigation, frame and detailed *layout* of a dialog can be characterized by separate UIPs. The *views* of a *window* can be structured by different UIPs on varying hierarchy levels. In this way, a modular structure of dialogs is enabled. In addition, versatile combinations, adaptability and extensibility of building blocks of a GUI will be promoted.

 ISBN: 978-1-61208-221-9

IV. EXPERIMENTAL APPLICATION OF UIPS IN GUI-MODEL-TRANSFORMATIONS

Up to now there have been no reports about experiences in the practical application of formal UIPs. The particular steps to be performed for a model-to-code-transformation and the shape as well as the outline of a formalization of UIPs have to be examined in detail. In order to gain further insights about UIPs, they have been experimentally applied by two different prototypes. Similar to the probing of software patterns, selected UIPs were instantiated for simple example dialogs. These are illustrated in Figure 4.

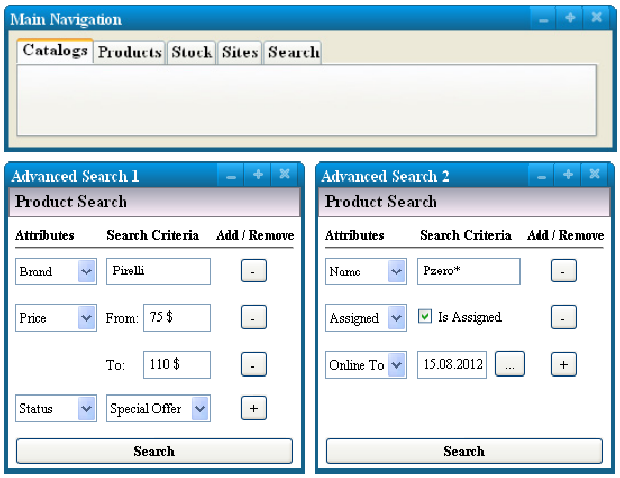

Figure 4. Example dialogs used for prototypes

On the one hand, the examples consisted of a *view* fixed in shape that contained the UIP „Main Navigation“ [12] on the upper part. On the other hand, the lower part shows two variants for a *view* whose visuals are dependent on the input of the user. Thereby, the UIP „Advanced Search“ [12] was applied. This UIP demands for a complex presentation control and is characteristic for eShops. Depending on the choice of the user, the *view* and *interactions* are altered. The search criteria can be changed, deleted and added as depicted in Figure 4 by two variants. Both example dialogs should have been realized by formalized UIPs and one prototype.

A. Generation at Design Time

Scope. Firstly, generating code for the GUI based on previously specified UIPs was probed. In general, the possibility to generate an executable GUI with the aid of UIPs had to be proven. The UIPs had to be completely defined at design time. Testing of the prototype had to be conducted after the GUI system was fully generated.

Approach. Foremost, the simple UIP *Main Navigation* was realized. This informally specified UIP was formalized after a language for formalization had been chosen. By means of a self developed generator, a model-to-code-transformation was performed to create an executable dialog. Subsequently, the complete GUI system was started without any manual adaptions to the code.

Choice of formalization language. A comparative study of UIML and UsiXML was conducted.

Regarding the structure of a GUI-specification, UsiXML proposes numerous models in order to separate the different information concerns *domain objects*, *tasks* and *user interface*. Not all the models were mandatory in terms of the example because no explicit *essential model* was given. On the contrary, UIML operates with few sections within one XML-document. This is because the UIML format was easier to handle and learn with respect to the simple example.

According to *UI-Controls*, UsiXML defines precisely which types of *UI-Controls* are available and what properties they can possess. An additional mapping model would have to be created in order to assign these elements to the entities of the target platform. In contrast, with UIML and its peer-section this mapping can easily be specified. The mapping to the GUI-framework can be altered afterwards without the need for changing the already defined UIPs. Moreover, UIML offers a more flexible definition of *UI-Controls* since custom *UI-Controls* can be declared in the structure-section as well as their properties in the style-section [22]. In addition, UIML provides templates for integration and reuse of already defined UIPs in other UIP formalizations.

Concerning *layout*, UsiXML uses special language elements to set up a GridBagLayout. UIML offers two variants: Firstly, it is possible to use containers as structuring elements along with their properties. The containers have information attached that governs the arrangement of their constituent parts. Secondly, UIML provides special tags that are committed for layout definition. UIML has a more flexible solution by defining *layouts* with containers that can be nested arbitrarily.

Related to behavior, both languages define own constructs. Nevertheless, complex behavior is difficult to master without clear guidelines for both. Concerning the examples, the behavior was limited to the technical presentation control within a *view*.

Choice of UIML. We decided to apply UIML for the example dialogs. Firstly, UIML is more compact in structure and enables a higher flexibility for shaping the formalization. Secondly, many of the language elements and models from UsiXML were not actually needed for the UIP „Main Navigation“. Thirdly, even the „Advanced Search“ example could not profit from the vast language range of UsiXML since all possible variants for search criteria could not have been formalized. At least UIML offered the possibility to rely on templates in order to define all possible lines of search criteria composed of simple UIPs. UsiXML turned out to be too complex for these simple UIPs. In addition, it was not clear whether UsiXML permits the reuse of already specified UIPs.

Realization of „Main Navigation“. Java Swing was chosen as target platform. For the peer-section we decided to map the elements of „Main Navigation“ to horizontal JButtons instead of tabs. In the formalization the mandatory parameters for number, order and naming of *UI-Controls* were specified. As result, the UIP was described concretely. The architecture was structured following the MVC-pattern [1]. The sections of UIML were assigned to components like in Figure 5.

 ISBN: 978-1-61208-221-9

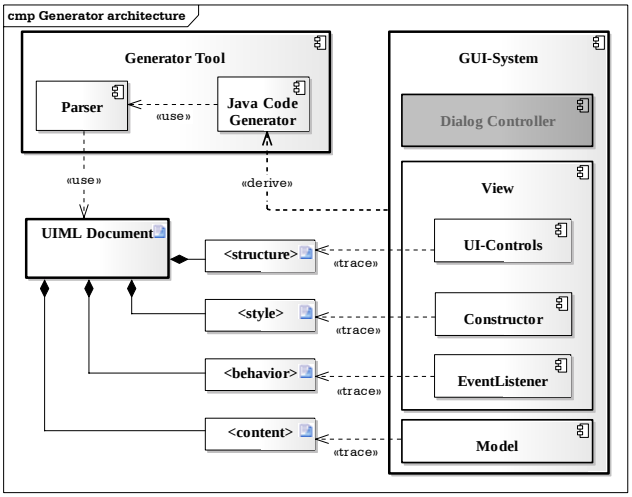

Figure 5. Architecture applied for code generation

Structure and style were processed within the object declarations (*UI-Controls*) of the *View* and its constructor. Based on the behavior-section, *EventListeners* were generated acting as presentation controllers. For the *Model* the content-section was assigned. Hence, the UIP "Main Navigation" formalized with UIML was transformed to source code.

Realization of „Advanced Search". Even by using the UIML templates, this complex dialog could not be realized by a generation at design time. It was not possible to instantiate the formalized UIPs that were depending on the choice of attributes at runtime.

Results. The prototype primarily was intended to prove feasibility. This is because we chose a simple architecture and did not incorporate a *Dialog Controller* for controlling the flow of dialogs. The control was restricted to the scope of the *UI-Controls* of the respective UIP. Thus, the behavior only covered simple actions like the deactivation of *UI-Controls* or changing the text of a label. Complex decisions during the interaction process like the further processing of input data and the navigation control amongst dialogs could not be implemented. A corresponding superordinate control could have been realized through a UIP-hierarchy in combination with appropriate guidelines for the formalization of control information. Despite the simplicity of the prototype, the following insights could be gathered:

Informal UIPs could be converted to formal UIPs by using UIML as a formal language. There was the need to define certain guidelines for this initial step. The *layout* of the example was specified by using containers for the main *window* and their properties. As a result, the *UI-Controls* were arranged according to these presets. Nested containers and complex *layouts* have not yet been used for the experiment in this way. The style also was described concretely within the UIML-document as well as the number and order of *UI-Controls*. The mapping of a formal UIP to a software pattern was simply performed by the scheme in Figure 5.

Concerning the example *Advanced Search*, only fixed variants or a default choice of criteria could have been formalized. The generator could have created static GUIs accordingly without realizing the actual dynamics of this particular UIP.

B. Generation at Runtime

Scope. The dynamic dialog *Advanced Search* could not be realized by the first approach. Thus, a solution had to be found that enables the instantiation of UIPs at runtime. Thereby, it was of importance to keep the platform independency of the UIML specification. The formal UIPs had to be processed directly during runtime without binding them to a certain GUI-framework.

Approach. Since the *Advanced Search* UIP was very versatile and could not be formalized with all its variants, the *layout* of the dialogs was fragmented. By the means of a superordinate UIP the framing *layout* of the *view* was specified in a fixed manner at design time. In detail, the headline, labels and the three-column structure of the *view* appropriate to a table with the rows of search criteria were defined.

The mandatory but unknown parameters that determine the current choice of criteria and UIPs had to be processed at runtime. Accordingly, a software pattern had to be chosen that is able to instantiate UIP representations along with their behavior. This pattern had to act similarly to the builder design pattern [10] which enables the creation and configuration of complex aggregates. In [23] a suitable software pattern was described which is explained shortly and illustrated in Figure 6:

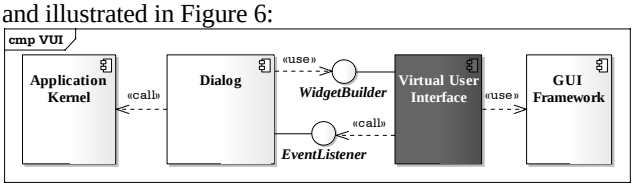

Figure 6. Virtual user interface architecture derived from [23]

Quasar VUI. The Virtual User Interface (VUI) of Quasar (quality software architecture) follows the intention of programming dialogs in a generic way. This means that the dialog and its events are implemented via the technical independent, abstract interfaces *WidgetBuilder* and *EventListener* rather than using certain interfaces and objects of a GUI-framework directly. By means of this concept, the GUI-framework is interchangeable without affecting existing dialog implementations. Solely the component *Virtual User Interface (VUI)* depends on technological changes. Upon such changes, its interfaces would have to be re-implemented. By using the interface *WidgetBuilder*, a dialog dynamically can adapt its *view* at runtime. For instance, the *Dialog* delegates the *VUI* to create and configure a new *window* containing certain *UI-Controls*. The *VUI* notifies the *Dialog* via the interface *EventListener* when events have been induced by *UI-Controls*. Both interfaces have to be standardized for a GUI system of a certain domain in order to enable the reuse of reoccurring functionality such as the building of *views* and association of *UI-Controls* with events

 ISBN: 978-1-61208-221-9

without regarding the certain technology or platform specifics being used.

VUI for UIPs. The concept, the *VUI* is based on, can be adapted to the requirements of the UIP *Advanced Search*. The idea is to instantiate complete *view* components with UIP definitions besides simple *UI-Controls*. The *Dialog* is implemented by using generic interfaces which enable the instantiation of UIPs, changing their *layout* and their association with events. In Figure 7 our refinement of the original *VUI* is presented.

The *VUI* for UIPs is based on our previously described generator solution. Each possible variation of *UI-Controls* matching the attributes of the domain *objects* for *Advanced Search* has been formalized before. Hence, the rows of the dialog were visualized by different UIPs. Concerning the formal UIPs, the proper implementations for the chosen GUI-framework were generated as stated in Section IV.A. The previously mentioned generator was integrated in the component *UIP Implementations*. These implementations of UIPs located within *VUI* are based on the interfaces and objects of the GUI-framework. In analogy to the *UI-Controls* already implemented in the GUI-framework, the available UIP instances were provided via the interface *UIPBuilder* and could be positioned with certain parameters.

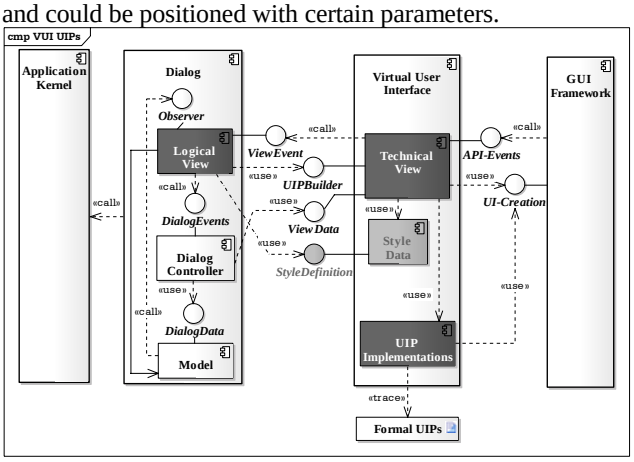

Figure 7. Virtual user interface architecture for UIPs

The *VUI* builds the *view* or a complete *window* as requested by the *Logical View*. Furthermore, the *VUI* provides information about the current composition and the *layout* of the *Dialog*. This information can be used by the *Logical View* for parameters to adapt the current *view* by delegating the *VUI* respectively. The *Dialog* coordinates the structuring of the *view* with the component *Logical View* and implements the application specific control in the *Dialog Controller* as well as dialog data in the *Model*.

Initially, events are reported to the *VUI* via *API-Events*. The *VUI* only forwards relevant events to the *Logical View*. When the respective event is only related to properties of a *UI-Control* or a UIP instance, it is directly processed by the *Logical View* which delegates the *VUI* when necessary. If the *Logical View* cannot process the particular event on its own, it will be forwarded to the *Dialog Controller*. For instance, this occurs when the user presses the button *Search* and a new *view* with the search results has to be loaded. The *Dialog Controller* collects the search criteria via the interface *ViewData* and sends an appropriate query to the *Application Kernel*. The result of the query will be stored as dialog data in the *Model*.

Results. For realizing *Advanced Search* with UIPs, a complex architecture had to be invented. Details like the connection of UIP instances to the *Dialog* data model as well as the automation potentials of the *Dialog Controller* could not be investigated extensively, yet.

The UIPs had to be specified in a concrete manner like in Section IV.A. The prototype was not mature enough to handle abstract UIP specifications. The style of the *UI-Controls* was also described concretely, so the control of style by a component of the *VUI*, as depicted in Figure 7, has not yet been realized.

Through the *VUI*, the versatile combinations of *Advanced Search* could be realized according to the example at runtime. The VUI constitutes of a component-oriented structure related to the software categories of Quasar [24]. Accordingly, it possesses its virtues like the division of application and technology, separation of concerns and encapsulation by interfaces. Despite its challenging complexity, a flexible and maintainable architecture for dynamic GUI systems has been created.

V. DISCUSSION

The theoretical reflection of the influence UIPs have on GUI transformations and the results of our experimental prototypes led us to the following findings.

A. Formalization of UIPs

Reflection of results. By experimentally probing the model-to-code-transformation of formal UIPs, we came to the conclusion that the generation of a GUI is not the complicated part of the process. Instead, the formalization and the occurring options in this step lead to the main problem. Primarily, the preconditions to benefit from the positive influences of the UIPs on the GUI development process have to be established by the formalization:

The generator solution was well suited for stereotype and statically defined UIML contents. In this context, *layout*, number and order as well as style of UIPs have been specified concretely. This led us to a static solution that can be applied at design time. But the UIP *Advanced Search* could not be realized by following this approach.

Parameters for UIPs. In order to overcome this static solution, a parameterization of formal UIPs has to be considered. Via parameters the number, order, ID, *layout* and style of *UI-Controls* within UIPs specifications have to be determined to provide a more flexible solution. Especially the number and order of *UI-Controls* have to be abstractly specified in the first place. In this way UIPs will be kept applicable for varying contexts. In place of a concrete declaration of style for each UIP, a global style template has to be kept in mind. By using this template, dialogs could be created with uniform visuals and deviations are avoided. For this purpose, the *VUI* incorporated the *Style Data* component. It is intended to configure the visuals of UIP

 ISBN: 978-1-61208-221-9

instances and *UI-Controls* globally. The configuration is used for the instantiation of these entities by the *Technical View*. Consequently, style information from single UIP specifications could be avoided and the UIPs would receive a more universal format.

B. Generation at Design Time

In principle, complex UIPs or UIP-hierarchies can be realized with the generation at design time. The easiest cases are elementary or invariant UIPs like calendar, fixed forms or message windows. These examples can be generated with ease since they do not need parameters besides a data model. For UIPs, which require parameters such as hierarchical UIP structures, an additional transformation is needed prior to the generation of source code:

Transformation of abstract UIPs. Firstly, the UIP is abstractly specified along with all parameter declarations needed and placeholders for nested UIPs. Subsequently, these parameters have to be specified via a context model which adapts the UIP to a certain application. Based on the abstract UIP specification and the context model, a model-to-model-transformation is performed in order to generate concrete UIP specifications like they were used in our examples. In this state all required information is available for the generation of the GUI system. The described model-to-code-transformation can be performed as a follow-up step. It has to be examined whether a suitable format is given to realize this approach, by means of UsiXML or IDEALXML and their models *AUIM* and *CUIM*.

C. Generation at Runtime

Regarding the UIP *Advanced Search*, it is clear that a large gap has to be bridged between the *essential model* and the *user interface*. A *use case* which demands for such dynamic UIPs hides a whole variety of different GUI-designs. Consequently, one static *user interface* cannot always be established for the elements of the *essential model*. However, even for these dynamic GUIs UIPs can serve as media to enable reuse of numerous aspects directly by generation along with a composition at runtime. The combined application of both our approaches can provide a feasible solution. Concerning the example from Figure 4, the previously generated *layouts* actually were reused for the *Advanced Search window* and the *views* of search criteria. By instantiation of matching UIPs, even the interactions respectively the presentation control was reused as well.

Generation of dialogs. As shown with our example, the current VUI is capable of the instantiation and composition of single parts of a certain *Logical View*. The generation of complete *Logical Views* on the basis of formal UIPs and their hierarchy could possibly be realized with the VUI architecture. The model describing the *Logical View* has to refer to the standardized interfaces of the VUI and a common UIP catalog. To formally specify the UIPs to be used in this environment, only UIML currently seems to be suitable. Firstly, an analysis of the required and reused elementary UIPs as well as the relevant *UI-Controls* has to be conducted in order to populate the basic level in the hierarchy of UIPs. Next, these UIPs have to be formalized with UIML along with their required data types and invariant behavior that acts as a basis for presentation control within the *VUI*. Furthermore, the interaction and *layout* within the *Logical View* have to be specified using UIML as well. This is because UIML already offers templates that can be parameterized and thus used for the composition of several UIP-documents into one master document establishing a UIP of higher level. Concerning UsiXML, one dialog can only be specified by a single AUIM respective CUIM document.

To complete the *Dialog*, meaning *Dialog Controller* and *Model*, relevant information on *tasks* and data *objects* has to be incorporated into a formal model. The research on the collaboration between adaptable UIPs and these logical aspects has just begun.

D. Limitations through the Application of UIPs

Individualization. Using UIPs instead of time-consuming manual transformations, a compromise is being contracted: A full individualization of the GUI is not possible with UIPs since the customizing is conducted within the limits of available and formalized UIPs. The UIPs can embody a further building block of standard software. Customization will be facilitated by defined parameters and automation.

Metamodels. The application of UIPs demands for clear guidelines for modeling of the *essential model* which result in a second limitation. The rules for this model need to define stereotype element types and their delimitations. The definition of the *essential model* is governed by a metamodel in the best case. Based on the metamodel, the elements can be defined uniformly and as stereotypes. For instance, it will be defined what types and refinements of *tasks*, *domain objects* and domain data types do exist in order to assign them homogenously to certain UIP categories. This concept is essential for the proposal of suitable UIPs for the partly automated development of GUI systems. The proposing system needs to work in two ways: On the one hand, the GUI developer asks for a suitable selection of UIPs for a certain part of the *essential model* at design time. On the other hand, users need to be provided with suitable UIPs in dynamic dialogs at runtime based on their current inputs. The mechanisms can only work if a uniform *essential model* with clear defined abstractions derived from fixed guidelines is available as fundamental information.

VI. Conclusion and Future Work

A. Conclusion

We theoretically and experimentally elaborated that UIPs do have numerous positive influences on the GUI development process. UIPs integrate well in the common GUI transformations. Therefore, our findings are not restricted to the domain of eShops but rather can be adapted to other standard software such as enterprise resource planning systems. Even for individual software systems, UIPs can be of interest in case that numerous GUI aspects are similar and their reuse appears reasonable.

Currently, adaptability and reuse of UIPs is limited to their invariant formalizations. UIP compositions could only

 ISBN: 978-1-61208-221-9

be created by manual implementation. We pointed to the limitations of current UIP specification format options and presented architectural solutions for their practical application. Above all, the upstream transformation of the abstract UIP description into UsiXML or UIML is worth to be considered since one could use their strength in concretely specifying user interfaces. Afterwards, the generation of GUIs based on this information would pose a minor issue.

B. Future Work

Formalization. For future work, we primarily see the research in formalizing UIPs. An important goal is to enable UIPs to act as real patterns that are adaptable to various contexts. The synthesis of a UIP-description model is the next step to determine properties and parameters of UIPs exactly and independently from GUI specification languages. Consequently, it can be more accurately assessed whether UIML or UsiXML are able to express the description model and thus UIPs completely. The independence from the platform can be achieved by both languages. However, it was not possible to specify context independent UIPs besides invariant or concrete UIPs. In this regard, the composition of UIPs, to form structured and modular specifications, remains unsolved, too.

Paradigm. Another open issue exists in the field of interaction paradigms [7] and the applicability of UIPs. With respect to the procedural paradigm, processes are defined which exactly define the single steps of a *use case* scenario. To provide a matching *user interface* for this case, additional information needs to be included in the formalization of UIPs. For instance, the process or *task* structures have to be specified by UIPs on a high level of hierarchy. These UIPs possess little visual content, maybe a framing *layout* for *windows*, and mainly act as entities for controlling the application flow. The *Dialog Controller* from Figure 5 and Figure 7 could be based on such a UIP. In this paper, no information for these components was integrated in the formal UIPs. So these components had to be implemented manually. For example, the *Dialog Controller* opens a new *window* with search results for the *Advanced Search*, controls the further navigation and delegates the structuring of the next or previous *windows*. In this context, our *VUI* solution is a compromise between automation and the reuse of elementary and invariant UIPs through manual configuration of the *Dialog Controller* and the delegated *Logical View*. A full automation needs further research.

REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stahl, A System of Patterns, New York: Wiley, 1996.

[2] M. Fowler. Patterns of Enterprise Application Architecture, Addison-Wesley, Boston, 2003.

[3] M. Haft, B. Olleck, “Komponentenbasierte Client-Architektur,” in Informatik Spektrum, 30(3), 2007, pp. 143-158, doi: 10.1007/s00287-007-0153-9

[4] M. van Welie, G. C. van der Veer, A. Eliëns, “Patterns as Tools for User Interface Design,”. in Tools for Working with Guidelines, Springer, London, Eds.: Ch. Farenc, J. Vanderdonckt, 2000, pp. 313-324.

[5] M. J. Mahemoff, L. J. Johnston, “Pattern languages for usability: an investigation of alternative approaches,” Proc. Computer Human Interaction, pp.25-30, 15-17 July 1998, doi: 10.1109/APCHI.1998.704138

[6] J. Vanderdonckt and F.M. Simarro, “Generative pattern-based Design of User Interfaces,” Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS '10), ACM, June 2012, pp. 12-19, doi: 10.1145/1824749.1824753.

[7] M. Ludolph, “Model-based User Interface Design: Successive Transformations of a Task/Object Model,” in User Interface Design: Bridging the Gap from User Requirements to Design, CRC Press, Boca Raton, Ed.: L.E. Wood, 1998, pp. 81-108.

[8] N. J. Nunes, “Representing User-Interface Patterns in UML,” in International Conference on Object-Oriented Information Systems (OOIS 2003), LNCS 2817, D. Konstantas, M. Léonard, Y. Pigneur, S. Patel, Eds. Heidelberg: Springer, 2003, pp. 142–151, doi: 10.1007/978-3-540-45242-3_14.

[9] A. Dearden and J. Finlay, “Pattern Languages in HCI; A critical Review,” Human-Computer Interaction, 21, 2006.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Reading: Addison-Wesley, 1995.

[11] S. Fincher, J. Finlay, S. Greene, L. Jones, P. Matchen, J. Thomas, and P. J. Molina, “Perspectives on HCI Patterns: Concepts and Tools (Introducing PLML),” Ext. Proc. Computer-Human Interaction (CHI’2003). Workshop Report, ACM Press, 2003, pp. 1044–1045.

[12] M. van Welie, “A pattern library for interaction design,” http://www.welie.com 10.05.2012.

[13] Open UI Pattern Library, http://www.patternry.com 10.05.2012.

[14] A. Toxboe, “User Interface Design Pattern Library,” http://www.ui-patterns.com 10.05.2012.

[15] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, and M. Florins, “UsiXML: a User Interface Description Language for Specifying multimodal User Interfaces,” Proc. W3C Workshop on Multimodal Interaction (WMI'2004), 19-20 July 2004.

[16] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, “UIML: An Appliance-Independent XML User Interface Language,” Proc. Eighth International World Wide Web Conference (WWW’8), Elsevier Science Pub., May 1999.

[17] R. Pawson and R. Matthews, Naked Objects, Chichester: John Wiley & Sons, 2002.

[18] H. Balzert, “From OOA to GUIs: The Janus system,” IEEE Software, 8(9), February 1996, pp. 43-47.

[19] L. Constantine, “The Emperor Has No Clothes: Naked Objects Meet the Interface”, http://www.foruse.com/articles 10.05.2012.

[20] D. Kulak, E. Guiney, Use Cases: Requirements in Context, New York: Addison-Wesley, ACM Press, 2000.

[21] K. Bittner, I. Spence, Use Case Modeling, New York: Addison-Wesley, 2003

[22] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml 10.05.2012.

[23] E. Denert, J. Siedersleben, „Wie baut man Informations-systeme? Überlegungen zur Standardarchitektur,“. in Informatik Spektrum, 23(4), 2000, pp. 247-257

[24] J. Siedersleben, Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar, 1st ed. 2004, corrected reprint, Heidelberg: dpunkt, 2006

 ISBN: 978-1-61208-221-9