An Analysis Model for Generative User Interface Patterns

Stefan Wendler, Detlef Streitferdt
Software Systems / Process Informatics Department
Ilmenau University of Technology
Ilmenau, Germany
{stefan.wendler, detlef.streitferdt}@tu-ilmenau.de

Abstract — **Graphical user interfaces (GUIs) are a crucial sub-system of current business information systems. They provide access for users to application kernel services in correspondence to business processes. As the processes and services change dynamically in our days, there is a strong need to adapt GUIs quickly to the changes. To enable both efficiency and usability during the adaptation, ongoing research has suggested to resort to model-based development processes, which employ patterns and their instantiation for specific GUI contexts. Those patterns are based on human computer interaction patterns and need to be formalized for their automated processing by generator tools. However, current research is still at the edge to express the concepts for such generative user interface patterns. The state of the art is not able to cover crucial factors of those patterns and misses a standardized format. Continuing our previous work on requirements for user interface patterns and their aspects, the aim of this paper is the development of an analysis model, which is able to express those needs in more detail using a semi-formal notation. With this step, a detailed description of generative user interface patterns is achieved, which can be the basis for the verification of current approaches of model- and pattern-based GUI development or even a deeper analysis.**

Keywords — user interface patterns; model-based user interface development; HCI patterns; graphical user interface.

I. INTRODUCTION

A. *Motivation*

Domain. Business information systems of our days are being maintained to upkeep or raise their effectiveness in supporting users carrying out operative tasks, which are demanded by the business processes of the respective company. Being a layer of a given business information system, the graphical user interface (GUI) is part of a value creation chain, as it enables the user to access functional, data and application flow related components of sub-systems located lower in hierarchy. Accordingly, the GUI allows the user to select and initiate functional behavior that processes data relevant to active tasks. As result, value is being created, which is meaningful to the sequence of the business process within the value creation chain. Due to systems are constantly matched closer to the set of tasks of the business processes and thus users are facing an increase in task scope and complexity, the need for well designed and adaptive GUIs has emerged.

GUI requirements. In this context, a user interface primarily is required to fulfill both the criteria of functionality and usability. On the one hand, a GUI has to reflect the current process definition and thus offer access to the respective activities in order to provide effective support for the user. On the other hand, for this support to be efficient, the non-functional requirement of usability, which embraces the suitability for the task and learning, as well as a high degree of self descriptiveness [1], plays an important role for testing and the acceptance for productive runs.

GUI adaptability. As business processes tend to change over time, the functional requirements based on them, such as use cases or task models, may change considerably, too. With those changes taking place, new requirements, having a significant impact on the GUI artifacts, are being introduced. Consequently, this part of the system has to conform to a high demand on adaptability besides the first release-specific requirements. Especially standard software systems, which offer a configurable core of functions to support business models, like applied in E-Commerce, see a distinctive demand for adaptive user interfaces [1]. Accordingly, a user interface of a business information system has to be based on a software architecture or development process, which facilitates the transition to new visual designs, dialogs, interaction designs and flows without causing significant costs in manpower and time.

Current limitations. Nowadays, the above mentioned requirements still cannot be accomplished fully by automation and generative development processes. On the one hand, available GUI-Generators can only cover certain stereotype parts of the user interface and may not lead to the desired quality in usability [1][3]. On the other hand, model-based development processes, which are able to generate more sophisticated user interfaces, also cannot support all variations on interaction and visual designs the changing business processes may demand for [4]. Finally, concepts that combine increased reuse and automation in user interface development and adaptation are being sought of.

User Interface Patterns. Together with other researchers [1][3][10][11][12][22], we believe that certain aspects of the GUI can be modeled independently in order to be composed and instantiated to their varying application contexts. As evolution and individualism in GUI implementations generally induce high efforts, an approach has to be followed, which enables a higher degree of reuse and hence allows for more common basic parts to be shared along components. For reuse, the basic layout of a dialog, its positioning of child elements and navigation flow as well as reoccurring user interface controls (UI-Controls) and their data type processing are to be mentioned as candidates for automated generation. In this context, the occurring variability needs to be expressed by new artifacts in the development process chain. The need for a systematic description of reusable GUI artifacts arose and initially has

 ISBN: 978-1-61208-276-9

found its expression in human computer interaction (HCI) [5][6][7] or, more recently, in user interface patterns (UIPs) [8][9]. In this regard, UIPs describe the common aspects of a GUI system in an abstract way and the developers concretize them with the required parameter information suited for the context of their instantiation.

UIP conception. The existing work about UIPs applied in model-based development processes [10][11][12] has laid down conceptual basics and milestones towards experimental proofing. However, no dedicated pattern definition for user interface development [14] has emerged yet and so, the motivation of the PEICS 2010 workshop still stands [15].

Factor model. To progress towards a more detailed and complete UIP conception, we deeply elaborated requirements with impacts to architecture, formalization and configuration of UIPs in [4]. A process, which enables the instantiation of UIPs and their compositions to form a GUI of high usability and adaptability, altogether, needs such a clear basis of requirements. However, the factors we have modeled, reside on a descriptive level that is not favorable to be directly translated to notations or formats for generative UIPs.

B. Objectives

The impacts of our factor model in [4] have led us to the strategy, to specify an analysis model for the UIP aspects and their various impacts. This model serves as a medium to close the gap between descriptive requirements of the factor model and formal notations. With the analysis model, we are detailing the requirements even more and progress towards a semi-formal notation for their description. The model is intended to capture all essential aspects, properties and required parameters for context-specific application of UIPs. With this contribution, a first version of the analysis model is presented.

In this regard, we focus on the UIP representation and not its mapping or deployment process, since other researchers have advanced in that area, but still lack a proper UIP representation. This representation is elaborated here along with related work, criteria, examples and finally an analysis model. The following questions shall be answered by our model:

- What information is needed to describe a UIP as a generative pattern applicable as a GUI architecture design unit?
- What elements a formal language has to feature in order to permit the full specification of such UIPs?

C. Structure of the Paper

The following section provides an overview of the pattern type to be covered in this work. Additionally, we summarize the outcomes of our previous work on the examination of model-based development processes and requirements related to UIPs. In Section III, the problem statement is formulated. This is followed by our approach in Section IV. The elaboration of the analysis model is presented in Section V. The results of our work are reflected in Section VI, before we conclude and suggest future work in Section VII.

II. Related Work

A. Human Computer Interaction Patterns and User Interface Pattern Definition

To open the discussion of reusable GUI entities, aspects of patterns related to GUI development are now introduced. We approach the term “user interface pattern” (UIP), which will drive the further elaboration of related work. For this purpose, we ask what the origins for definitions of UIPs in the context of UI generation are.

HCI pattern ambitions. The early stages of patterns for user interfaces were determined by the goal to describe reoccurring problems and feasible solutions for GUI design offering high usability. Borchers [7] stated that human computer interaction (HCI) experts had a hard time communicating their feats in ensuring a good design of a systems GUI to software engineers. Thus the idea was born to express good usability via patterns as this was already a good practice for software architecture design. In this regard, Van Welie et al. [16] argued that patterns are more useful than guidelines for GUI design. In addition, they suggested the term pattern for user interface design along with criteria how to assess the impact on usability of each pattern.

Research into HCI patterns went on and culminated into pattern languages such as the one created by Tidwell [17]. Prior to this development, Mahemof and Johnston [5] outlined a hierarchy of patterns, what already implicated that there are complex relationships inside HCI pattern languages.

No unified pattern notation. Some years later, Hennipman et al. [18] claimed that available HCI pattern approaches could be improved as there are still obstacles for their efficient usage. Their analysis of relevant sources reveals major issues such as the missing guidelines how to formulate new HCI patterns, integrate them in tools and how to apply them. The request for a standard pattern specification template already was formulated by [16] and [7]. In this regard, Borchers mentions early sources adopting the pattern notion by Christopher Alexander. Thus, Fincher finally introduced PLML [19] in [20]. However, the issue of a missing standardized pattern format still persists [15], which eventually is detailed by Engel et al. [21]. Therein, they analyze the shortcomings of current HCI pattern catalogs and the intended standard notation of PLML.

UIP definition. Vanderdonckt and Simarro [22] separate two main representations of patterns based on the intended usage. Descriptive patterns serve a problem description and solution specification purpose. In contrast, generative patterns feature a machine readable format as they are to be processed by tools and in particular GUI generators.

B. Formal Languages for GUI Specification

Now, we ask if there are languages available that permit the formal specification of GUIs or even UIPs.

In our previous work [1][8], we already went into the possibilities to express UIPs with the means of mature GUI specification languages UIML [23] and UsiXML [25]. As these languages are focused on platform-independent full-fledged GUI specification and intended to be machine processed, some of their elements may be candidates to be

included in a sophisticated UIP definition model. Both languages feature common elements to define the visual layout, interactive behavior, and content of a certain GUI part. For pattern-specific application UIML and UsiXML differ in their capabilities: UIML incorporates elements for template definition and a peer section, which decouples structures or UI-Controls within the layout from their technical counterparts. In contrast, UsiXML is based on a more complex approach, which defines a metamodel consisting of a model hierarchy and methodology [26]. The abstract and concrete user interface model may be of relevance for our objective.

C. Influence Factor Model for User Interface Patterns

Continuing on previous work, we progressed towards an elaborate influence factor model for UIPs, which is depicted in Figure 1. Motivated by missing standards and competing UIP notations inside modeling frameworks, this model was intended to establish an independent requirements view on the formalization and instantiation of generative UIPs: We took our examples and architecture experiments [1], as well as criteria, aspects and variability concerns [8], and refined them. The requirements stand close to the profile of current approaches in research. For details, [4] can be consulted.

The UIP definition to be sought after has to introduce a pattern conception, which is backed by a limited set of types, roles, relationships and collaborations among GUI related specifications and components. Because of the complex nature of both GUI architectures and specifications, a restriction and specialization of the entities to be involved in the development environments for pattern-based GUIs have to be set. Along with this restraint, the GUI specific kind of pattern still needs to be abstract in order to enable vast customization and instantiation to differing contexts. The major share of the patterns vigor has to be sourced from the similarity in structural (*view aspect*) and behavioral (*interaction* and *control aspect*) definition of new GUI entities.

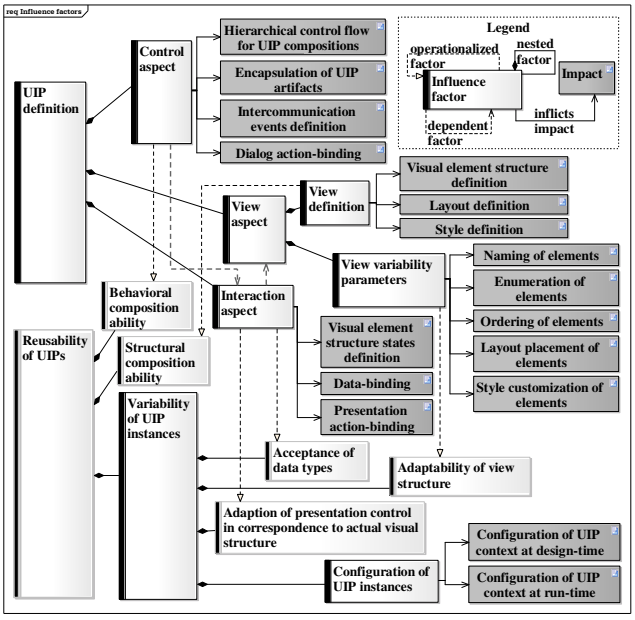

Figure 1. Influence factor model for generative UIPs described in [4]

In other words, the pattern definition introduces certain quality aspects in GUI design, which can be altered quantitatively, when they are respectively complemented with necessary structure, layout and style details (*view variability parameters*) as well as combined with each other (*behavioral* and *structural composition abilities*). This commonality ensures that no longer specialized solutions or manually refined structures, which cannot be covered by mere UIP instantiation, are applied in the same GUI system architecture.

D. Model-Based Development Processes involving User Interface Patterns

The enhancement of model-based development by generative UIPs already found strong reception. In reference [4], we presented an overview and assessment of the approaches of Zhao et al. [1], PIM [27], UsiPXML [10], PaMGIS [11] and Seissler et al. [12]. For a summary, Table I TABLE Icompares the above described approaches.

TABLE I. COMPARISON OF APPROACHES FOR MODEL-BASED DEVELOPMENT EMPLOYING UIPS

	Approach			
	Zhao et al.	***UsiPXML***	***PaMGIS***	***Seissler et al.***
Pattern types	Task patterns based on [28], set of window and dialog navigation types	Task, dialog, layout and presentation	Task and presentation patterns, fine grained hierarchy based on	Task, dialog and presentation patterns
UIP formal-ization notation	Unknown	Enhanced UsiXML	Unknown, XML based, <automation> tag and DTD	Embedded UIML supplemented by parameter and XSLT enhancements
UIP config-uration	At design	At design	At design	At design and run-time
Process output	Target code	UsiXML, M6C	Target code	Augmented UIML to be interpreted

Not all of the factors' impacts were supported or inspired by the approaches. A summary of realized (arrow in a box) or inspired (single arrow) impacts is given by Figure 2.

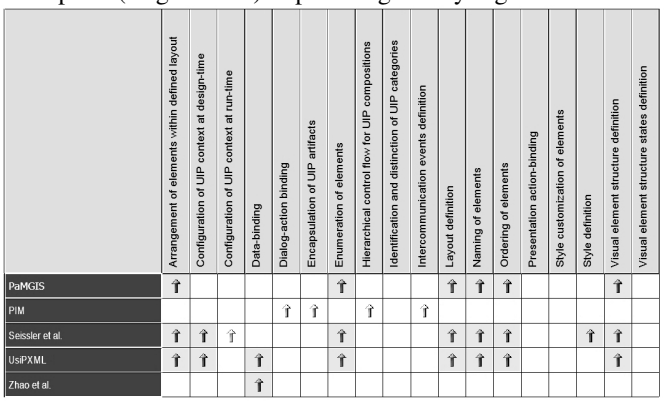

Figure 2. Impacts covered by examined approaches

Since our valuation revealed that there were many open issues associated with the different approaches, we only considered the full and no partly or probable realization of an impact. Notably is that the *view aspect* was realized by the most recent approaches. In contrast, the *interaction aspect* was only considered for *Data-binding*. Moreover, the *control aspect* was not realized by any approach, but inspired by PIM. Lastly, the *Configuration of UIP instances* was restricted to design-time only, but already inspired by Seissler et al. in reference [13].

III. PROBLEM STATEMENT

A. UIP Definition

Descriptive UIPs. From our observations concerning descriptive UIPs, we learned that they are well-understood as specification elements and supported by the HCI community. Nevertheless, the research into descriptive HCI patterns has not yet converged towards a standardization for the structure and organization of UIPs [15][21].

Generative UIPs. Generative UIPs may be classified as software patterns and as those they need a formal notation, and thus, are seldom encountered.

From our point of view, the past work on HCI patterns is concentrated on the descriptive form. As there is no unified approach in specification and usage of descriptive HCI patterns, they can hardly be used to source and abstract common elements of a generative representation. First and foremost, descriptive UIP sources may be a useful resource to assemble dialogs that may act as representative examples for a certain system or domain. On that basis, requirements or criteria for UIP formalization can be inductively obtained. Partly, we revert to this approach and sketch some example UIP instances in Section IV.B.

As a consequence, there is a large gap concerning the detailed definition of generative UIPs. Thus, a format for UIPs has to be found that is at least able to express most impacts of view and interaction aspect. Filling the gap with their own UIP concepts and notations, the model-based approaches of Section II.D are converging concerning the view aspect, but failed to convey all UIP impacts.

B. Formal GUI Languages and model-based Development

Enhancements. As there is still no dedicated language for UIP formalization, developers have to revert to existing GUI specification languages like UIML or UsiXML, which will be referred as XML languages in the following. As a result, two factions among the model-based approaches arose, one using UsiXML and the other applying UIML. Both languages need enhancements to express UIP related variability. Accordingly, the approaches incorporated their own parameter and configuration concepts. In sum, they all failed to publish enhancements that empower the specification languages regarding the interaction and control aspects. Currently, the notations are restricted to the view aspect mostly.

Generation of XML specifications. The XML languages have been developed to offer a platform-independent specification of GUI systems. In this context, they have been based on a metamodel that is somewhat similar to common universal object-oriented programming languages, which cannot handle aspects or traits and thus are incapable of expressing patterns in their abstract form. The XML languages clearly fail in the fulfillment of the reusability, variability and composition ability criteria [8].

However, applying the XML languages for their original purpose, apart from pattern definition, may play out their strengths. Accordingly, developers could use them for concrete GUI definition and final rendering to the desired platform. To integrate UIPs in this procedure, a generation of XML language code could be a possible solution to overcome the inabilities as proposed in [1]. This idea was already followed either by generation of UsiXML [10] or the interpretation of UIML [12]. The XML code would hold the already instantiated UIPs or the required information for rendering. The benefit would be the possibility to use existing tools for the XML languages. In addition, a more important merit would exist in obtaining a concrete user interface level (CUI) specification [26], and thus, the ability to be independent from platform specifics.

In any case, a new language or extensions for the XML languages are to be sought after. Whether UIPs are being defined concretely in XML or the latter is generated, the XML languages will be a fundamental part of this solution. Consequently, the new language must facilitate the expression of UIP instances in rich XML language specifications. For that purpose, a unified UIP-model has to be established, which truly holds all information for the definition of generative UIPs and parameters for their transformation to UIP instances or instance compositions forming a concrete GUI model.

IV. OUR APPROACH

A. Strategy

As mentioned in the objectives, the impacts in reference [4] resulted in the strategy to develop an analysis model, which is aimed at further detailing the UIP aspects. We develop a structural model that is biased towards an implementation of a dedicated UIP language.

Motivation of an analysis model. Some requirements such as *interaction* and *control aspects* are cross-cutting concerns and are really hard to achieve for pattern formalization. Thus, more planning and rationale is required before we can consider the development of a dedicated language. We follow the way of traditional modeling of requirements and ease their transformation to design with an analysis model. The model is intended to express the domain terms and concepts with a structure.

With a structural and more detailed model, the tracing of the influence factor impacts to potential solutions is better possible than with the pure influence factor model presented by Figure 1. In the factor model, there exist no separated entities that are modeled with their attributes and relationships to reflect a possible solution approach.

Assessment of recent approaches. Although we pointed out the factor support and issues we could so far discover as result of our assessment of other available approaches in reference [4], we also concluded that more details on examples and the applied notation have to be revealed in order to refine the assessment. By developing an analysis model, we seek to overcome the lack of detail and rationale

 ISBN: 978-1-61208-276-9

on the design of notations suitable for UIPs. The notation to be used for modeling is the UML 2.0 class model.

Why do we propose a semi-formal model? For a technical architecture design or a generative process for formal UIPs to be verified, a wide range of requirements emerging from the initial criteria have to be taken into account, which cannot comprehensively modeled on a formal basis. In contrast to other researchers directly pushing towards a formalization of UIPs, we think this intermediate step is necessary and helpful. In our opinion, a semi-formal model is more useful to the developer than a formal model in first place, hence the mental conception about full scale generative UIPs has to be inspired first. The understanding of these complex patterns, their aspects and element relationships is the primary goal that should not be hindered by formal media, which cannot be imagined easily. A semi-formal model enables a better understanding than a grammar, since it may visualize concepts, their structure and relations depending on the chosen notation.

In sum, the model has to satisfy the information needs of the developers first, before they can think of how to employ the available formalization options or even GUI XML languages to express the requirements residing inside the model. Primarily, the model has to capture requirements in way that is easily understandable for human-beings.

Why do we apply the UML 2.0 class model? The UML class model lies in between the descriptive nature of the factor impacts and a formal notation. In this regard, a class model is already inclined towards a formal implementation. This is the case for class models serving as a design model for object oriented programming languages. In analogy, our analysis model may lead to a design for new language elements for the definition of generative UIPs. The language to be sought after also should rely on a structural paradigm, since the GUI implementations form a structure as well.

Moreover, a class model already proved useful for the expression of design patterns. The paradigm employed allows us to model abstract data types, their common attributes as well as their cardinalities and relationships. As the model entities all reside on an abstract level and do not describe already instantiated objects, the class model proves to be suitable for our task. More precisely, the UIP concepts can be modeled from a point of view where the abstraction and instantiation are separated. The class model forces the developer to express his solutions by abstractions that concentrate the commonalities of later instantiated objects. As we seek to express UIPs that feature reusable GUI solution aspects, a class model may provide a proper notation.

With the class model, we will be probing the modeling of required information for UIPs. Currently, developing a particular language or focussing on a certain architecture experiment seems to be too specific. In contrast, we investigate how the information of UIPs and their configuration can be established in general. To sort out possible options, trace factor impacts on more detailed granularity and map them to the final solution, the analysis class model may prove as a valuable asset. Finally, we may draft a coupling between a UIP, its configuration and GUI architecture or at least mandatory prerequisites.

B. User Interface Pattern Examples

By reason that we do not want to claim being able to establish a UIP analysis model applicable for each domain, we stick to business information systems as mentioned in the introduction. More precisely, as stated in Section III.A, we rely on common dialogs for E-Commerce applications as a basis. In fact, we subsequently derive the analysis model by focusing both on the factor model in Figure 1 and the following example dialogs.

Simple search. For an easy example, we start with a dialog that has the "Search Box" [28] pattern instantiated. The simple search illustrated in Figure 3 is mainly composed by a single panel (*ContentPanel*), which defines a GridBagLayout as seen in the upper part of Figure 3. The UI-Controls are fixed and aligned in respective fashion. For variability, only the concrete object data types need to be bound to the combobox and textfield. In fact, this kind of UIP is mainly invariant.

Advanced search. The next example shall be more complicated and thus, demand for every aspect described within the factor model. We decided for an "Advanced Search" [28] pattern, which alters its visuals and interaction options depending on user input.

Our example, depicted in Figure 4, mainly consists of two panels for layout definition as shown on the upper half. The panel *RootPanel* defines a GridBagLayout consisting of three cells (grey borders). Located in the center of this container, the *SearchCriteriaPanel* defines a layout of several rows each containing on cell (solid black borders). Additionally, the latter may grow or shrink in height to accommodate or discard search criteria lines to fit inside the container. Lastly, the *SearchCriterionPanel* (dashed borders) defines a layout appropriate for individual search criterions.

The usage of this dialog is as follows: Firstly, the user selects an object to be searched from the "Type of Object" combobox. Secondly, he chooses an attribute from the combobox inside the *SearchCriteriaPanel*. Accordingly, the UIP dynamically has to instantiate new sub-UIPs, which resemble the single search criteria rows. For each datatype, a pre-defined UIP, which is similar in shape to the *SearchCriterionPanel*, is assumed to be available. In the example, the datatypes String, price, and week are considered. With the buttons on the right hand side, the user may add or drop new search criteria rows and so the view aspect will change.

The variability is limited to the object types and their attributes to be searched with this UIP. Controller related aspects have to be adapted based on the UIP definition.

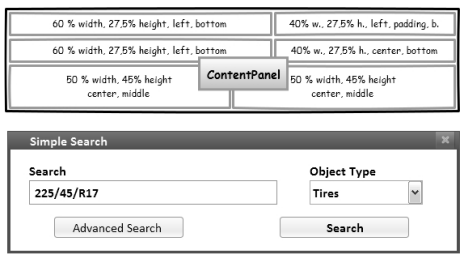

Figure 3. Simple search UIP example layout and dialog

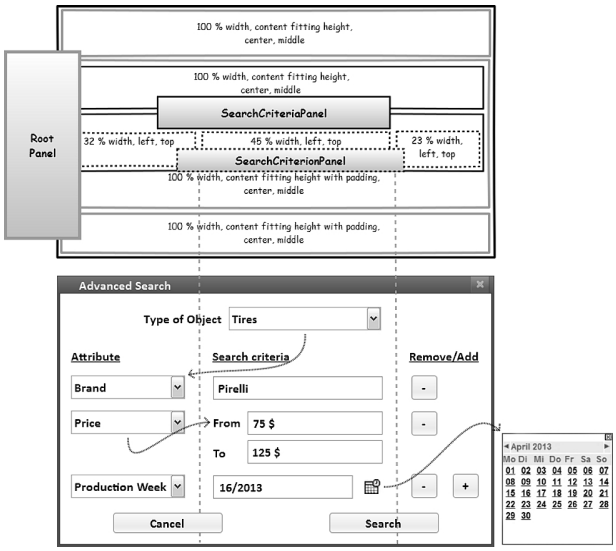

Figure 4. Advanced search UIP example layout and dialog

V. THE ANALYSIS MODEL

In this section, we develop the proposed analysis model. At first, we review each UIP aspect and its associated impacts in order to elaborate the decisions in design of the new model. Afterwards we present the structure of the model and finally apply the model to both examples introduced in Section IV.B. The terms in *italics* refer to respective analysis model elements.

A. Analysis Model Bias

On principle, there are two options on how to bias the model. Firstly, the model could be biased towards the software architecture and thus employ proven design patterns in its structures. This option would be rather suitable for generators and the further automated processing of the model, but it would be tedious to translate it back to the UIP requirements for the developers. In addition, the formal XML GUI languages (Section II.B) were not designed to accommodate architectural knowledge.

Secondly, the analysis model may be biased towards requirements and thus acting as a traditional analysis model, which captures and visualizes requirements. This option would be rather easy for the developers to understand, but would be costly to be translated to formal languages and generators. However, the translation to the XML languages is only a theoretical aspect, since generative UIPs cannot be expressed by their facilities as discussed in Section III.B. Eventually, we decided for the latter option.

B. General Rationale

Separation of definition and instances. A fundamental decision was the separation of elements or features that may be available in a UIP definition and the several element instances that may appear in a particular UIP instance for a certain context. In other words, we divided the UIP analysis model into two parts. One part holds the definition and reoccurring features (class names in black). The other part allows the description of instance information (class names in white).

UIP configuration. Following this approach, the main class *UserInterfacePattern* takes part in relationships that mostly focus on definition purposes, but also is connected to *UIPConfiguration*, which enables the description of particular UIP instances of the respective kind. The information used for pattern definition purposes will be covered in the following sub-sections. The configuration of UIP instances further branches into *Defaults* and *Parameters*. Both classes resemble containers that hold the *UIControl* instances, which are declared as *UIControlConfigurations*, for a particular UIP instance.

The *Defaults* are intended to omit stereotype configurations of default *UIControl* instances, which commonly appear in most contexts and shall not be defined redundantly. Concerning the example dialogs, the basic or invariant *UIControls* needed for user understanding and interaction like the labels, textfield and combobox of the simple search should be defined as *Defaults*, as there is hardly variability. This way, already established configurations may partly be reused among individual UIP instances. That means a UIP may contain pre-configured elements and parameters to avoid repetition. Later on, this facility will become useful for the dynamic adaptation of a UIP instance at run-time.

Both *UIPConfiguration* and *UIControlConfiguration* are primarily used for the "Configuration at design-time" impact and thus contain the declarations a developer may define in interaction with an "instantiation wizard" [10]. The configuration of *UserInterfacePatterns* and *UIControls* has to be separated, since both offer different sets of attributes, and more important, impact the GUI on different levels of abstraction or scope.

C. View Aspect Design

View definition. To begin with "View definition", this factor defines the *UIControls* or *UserInterfacePatterns* to be generally contained in a UIP specification unit as visual components. Both resemble a *ViewStructureElement*, which has a unique *ID* as identifier inside the pattern used by *UIPConfiguration* and *UIControlConfiguration* to reference the respective element. *UIControl* is a classifier for the various visual components or widgets a GUI framework may possess as types.

A UIP is always composed of a *ViewStructureElement* set and thus may build a varying hierarchical structure of those graphical elements. However, *ViewStructure* only holds each *ViewStructureElement* to be available to build instances once. The resulting element structure of a particular UIP instance is not described by *ViewStructure*. Instead, this is the responsibility of the configuration classes. The *ViewStructure* only defines what elements are generally available for the particular UIP. Based on that decision, the *ViewStructureElements* later may be exchanged without altering the already defined configurations.

For each *UIControl* of the resulting *ViewStructure*, style and general layout have to be defined. The style impact is not detailed here, since we have not came to a result in this regard and focused on the other impacts. For the sake of uniform views and maintaining corporate design, style information may be governed globally and locally by each

 ISBN: 978-1-61208-276-9

individual *UIPConfiguration*. In addition, there may be constraints for each element, which determine its allowed minimum and maximum occurrences.

Layout rationale. With respect to "Layout definition" impact, we ask if there is a need for dedicated layout-patterns or if the distinction between primitives (*UIControl*) and composites (*UserInterfacePattern*) is adequate.

Referring to UsiPXML [11], layout patterns can be defined separated from presentation patterns. How they are integrated at various stages in the hierarchy, and more important, how they can be handled dynamically at run-time, remains an open issue, as there were no detailed examples for pattern composition and specification code given.

In addition, it is arguable whether a layout is assigned separately to a paralleled UIP composition or if each UIP models layout partly but explicitly. Partly means that UIPs need to define attributes for the number of rows and columns of a grid, their relative width and height, as well as the alignment. A visual impression of the abstract layout definition expressed by UIPs is depicted in the upper parts of Figure 3 and Figure 4. We decided to model this information by UIPs, as for advanced search, the layout needs to be re-configured dynamically with respect to *SearchCriteriaPanel*. This panel may grow and shrink in row numbers.

Layout definition. Inspired by our examples, we treat the layout container as a UIP, and thus, a layout pattern is already merged inside. So, the above mentioned layout definition parameters have to be associated to each *ID* of a UIP-type class, since it is acting as a superior container. Consequently, the advanced search dialog consists of three UIPs designated as containers in Figure 4. Translated to GUI frameworks, this implicates that each UIP will be treated as a panel or even window frame with a certain *LayoutManager* attached. We reason our approach with the fact that every dialog at some stage needs layout containers and these are eventually to be mapped to peers in the GUI framework. The detailed parameters for layout, such as padding, orientation and size policies, may be governed globally.

View variability parameters. To configure parameters for an element of the *ViewStructure*, regardless of what type, the respective *ID* of that element is used as a reference.

The *UIControlConfigurations* assigned to UIPs influence the instantiated unit in a global way. So, for the *view aspect* the general layout of the instances *ViewStructure* is declared by *LayoutManager*, which decides on the actual grid, for example. This way, the layout and orientation of UIP instances may be altered, but have to be declared explicitly for each *UIPConfiguration*.

As the elements defined by a UIP are abstract, the reference to the *ID* acts in analogy to the class concept for object-orientation. In fact, the element occurrence is determined by the number of respective configurations. For the individual element instances, one or many *UIControlConfigurations* can be declared to specify their characteristics. More precisely, as *view aspect* parameters we arranged for *Name*, *Caption*, *and Order* inside a layout grid cell and *Style* of each element. Some of these parameters are even optional. With *LayoutPosition* the position of the element with respect to the declared *LayoutManager* can be defined.

D. Interaction Aspect Design

In the factor model, the *interaction aspect* was not separated between stereotype definitions and parameters, as this was done for *view aspect*. Finally, the main classes, which model the *interaction aspect*, resemble parameter types. Since the factors apart from the *view aspect* ones mostly resemble cross-cutting concerns, the resulting *interaction* and *control impacts* refer to the static and variable declaration of *view impact* elements as a basis. In detail, the *interaction* related *UIControlConfiguration* parameters comprise of *DataType*, *PresentationEvent* and *EventContext* as an additional child of the latter.

Coupling points. For a UIP definition to be integrated in a GUI architecture, there is the need to arrange for coupling points. These points allow the integration of automated generated code and manually defined UIP information. Potentially, these can be comprised of the following:

- Standard events (*control* - "intercommunication events definition", "dialog action-binding")
- Input and output data (*interaction* - "data binding")

The latter point may resemble GUI architecture models discovered in common MVC architectures. The mentioned coupling points are either evaluated (events) or processed by the dialog kernel or logic part of the dialog. It is not necessary for that component to know where data changes and events have originated from. So, these suggested coupling points may be a good starting point. Accordingly, events (*PresentationEvents* and *OutputActions*) and the "GUI Data Model" have been included in the analysis model.

Data-binding. The binding of a *UIControl* to certain data is accomplished by a *UIControlConfiguration* parameter. So, the *DataType* binds the elements to certain data structures. As *DomainDataTypes* may significantly differ from the types used by the GUI framework, the class *GUIProjection* is rather associated as the configured *DataType*. For the *DataType*, it can be configured if the data is to be displayed only (input) or if the user may conduct changes (output), which are finally applied to the GUI *Model* part. The *DataType* parameter also may be associated to *EventContext*, which configures the data to be submitted by a *PresentationEvent* of the respective element.

Besides the distinction between input and output, *Models* have to be provided as coupling points for both cases to obtain data for display. The application kernel has to provide a respective query to obtain *Entity* data and the GUI architecture has implement a certain *Model* to enable the presentation of the query with appropriate data types for *UIControls*, e.g., data conversion to strings or string lists. In this regard, aspects like the timing, refresh rate, lazy loading are no concern of the UIP definition and have to be implemented by the data sources or queries. The *Model* has to rely on the data source and is not responsible of those technical aspects. In contrast, the *Model* needs to provide the navigation inside data structures and the structuring of data for presentation purposes that may be altered from application and data layer designs in order to offer a suitable projection for human processing.

 ISBN: 978-1-61208-276-9

Currently, we are unsure how UIPs specific *Model* requirements are to be formalized. However, this information is essential for the coupling. In addition, it will provide useful for the checking of the validity of configuration and *view* variability of the UIP instance. Concerning the advanced search, there must be a *Model* available to provide object types and their attributes as well as another *Model* to accommodate the chosen search criteria as the dialog result.

Events rationale. For *PresentationEvents*, we enumerated some typical events implemented in GUI frameworks. To progress towards a unified solution for generative UIPs, we think that a standardization of events, *PresentationEvent* as well as *OutputAction*, and similar types is necessary. The integrative and strict type definitions of the GUI specification language UsiXML on CUI level [26] may be a valuable resource for that approach. Otherwise, both specification and tool processing would demand for niche solutions that are hardly manageable with respect to versions and dependencies. We wonder how UsiPXML [10] or the UIML UIP definition by Seissler et al. [12] are defined as a language to be integrated in tool environments, which are to handle the generic concept of their variables and assignments effectively. We have to wait for them to publish detailed language definitions and code examples.

Presentation action-binding. To bind an element to a certain *PresentationEvent* type, the desired event has to be included in the appropriate *UIControlConfiguration*. This event may be declared for various purposes concerning visual structure states as described below.

Visual element structure states definition. The first *interaction aspect* impact needs to be further detailed. Depending on the actual structure of the UIP, states that occur within the scope of the contained *UIControls* and states, which alter the view of embedded UIPs have to be covered. To trigger changes in state for both cases, only *UIControls* can be specified as sender of respective events.

***UIControl* states.** For changes in state, we consider the activation or deactivation as well as hiding and unhiding of single *UIControls* or sets of them. Those abstract events are to be translated to technical representations and their detailed implementation. For instance, a checkbox in a sub-form may deactivate the delivery address (if it is equivalent to billing address) or in another case, a collapsible panel may be collapsed. In our model, the *ViewStateAction* is defined as an abstract feature for a UIP. By the UIP specification, the possible actions are defined and associated to affected *UIControlConfigurations* and thus *UIControl* instances. Finally, for these actions triggering *PresentationEvents* can be associated.

Embedded UIP states. Since the possible states for composite UIPs cannot be enumerated or state machines finitely defined inside pattern specifications, we employ information, which describes the results of the state change, and thus, enables a generator to build appropriate state machines or comparative implementations.

The *ViewStructureAction* is designed to handle the change of visual states for UIPs. For the trigger, a respective *UIControlConfiguration* is needed, which is aimed at a certain *ID* to allocate the *UIControl* and the type of *PresentationEvent*. We considered the addition, replacement, or removal of UIP instances. This behavior is closely related to the <restructure> tag of UIML [24] and may be refined based on its semantics. However, for UIML these facilities can only be applied with already instantiated UIPs.

DynamicStructures are used for the addition, removal or replacement of *UserInterfacePattern* instances. They are selected on the basis of defined *Keys*, which enumerate certain *DataTypes* or *EventContext* data to assign pre-configured *UIPConfigurations* to the triggered *ViewStructureAction*. A *UIPConfiguration* may be used by more than on *Key*, which models a certain context situation. Concerning the advanced Search example, the *Model* holding the object and attributes lists must return values that match the specified keys. Each time a combobox is changed, the presentation event handling routine must query the *Model* for the selected objects attribute and its kind or type of representation. The query result will be embedded in the *EventContext*, which is matched to a *Key* value. So, the UIP and its *DynamicStructures* are based on a canonical representation of *DomainDataTypes*.

Moreover, the *ViewStructureActions* rely on pre-configured elements, which may only allow for variability concerning the *DataType*. They either rely on a self-reference (removal, replace) or additionally are associated to available elements of the *ViewStructure* (add, replace) via *DynamicStructures.*

However, this mechanism only makes sense for *UserInterfacePatterns*, which are specified by *Defaults* and always represented by default *IDs* present inside the *ViewStructure* of a UIP definition. In this way, the *DynamicStructures* will only affect default or invariant *UserInterfacePatterns* inside the given *ViewStructure*, hence it is not desirable to replace entire sets of UIP instances defined on behalf of the developer for a specific context. Thus, manually defined UIPs portions have to be separated from *DynamicStructures*.

Based on the considerations for *DynamicStructures*, we decided to associate *DataType* with *GUIProjection* rather than with *DomainDataType*. A reference to *DomainDataTypes* would have meant to define a *Key* and appropriate *UIPConfigurations* for each *DomainDataType.* Each change of types would have cascaded to each UIP relying on *DynamicStructures.* We believe that *GUIProjections* may be more stable than *DomainDataTypes* and even be shared among *DomainDataTypes.*

E. Control Aspect Design

Dialog action-binding. So far, we have not progressed to feasible results for most *control aspects*. Only the binding of *UIControls* to application actions has been included. Via the global *OutputAction* parameter declaration of a UIP, one can define what events of that kind are raised by the *UIControlConfigurations*. These can be bound to a certain *UIControl* only by a link with the *PresentationEvent*.

F. Structure View on the Analysis Model

The resulting analysis model is illustrated by Figure 5. The classes shaded in medium grey are related to the "view definition" factor. Configuration related classes are shaded in dark grey and feature a white caption. Most *interaction aspect* impacts are supported by the classes shaded in white.

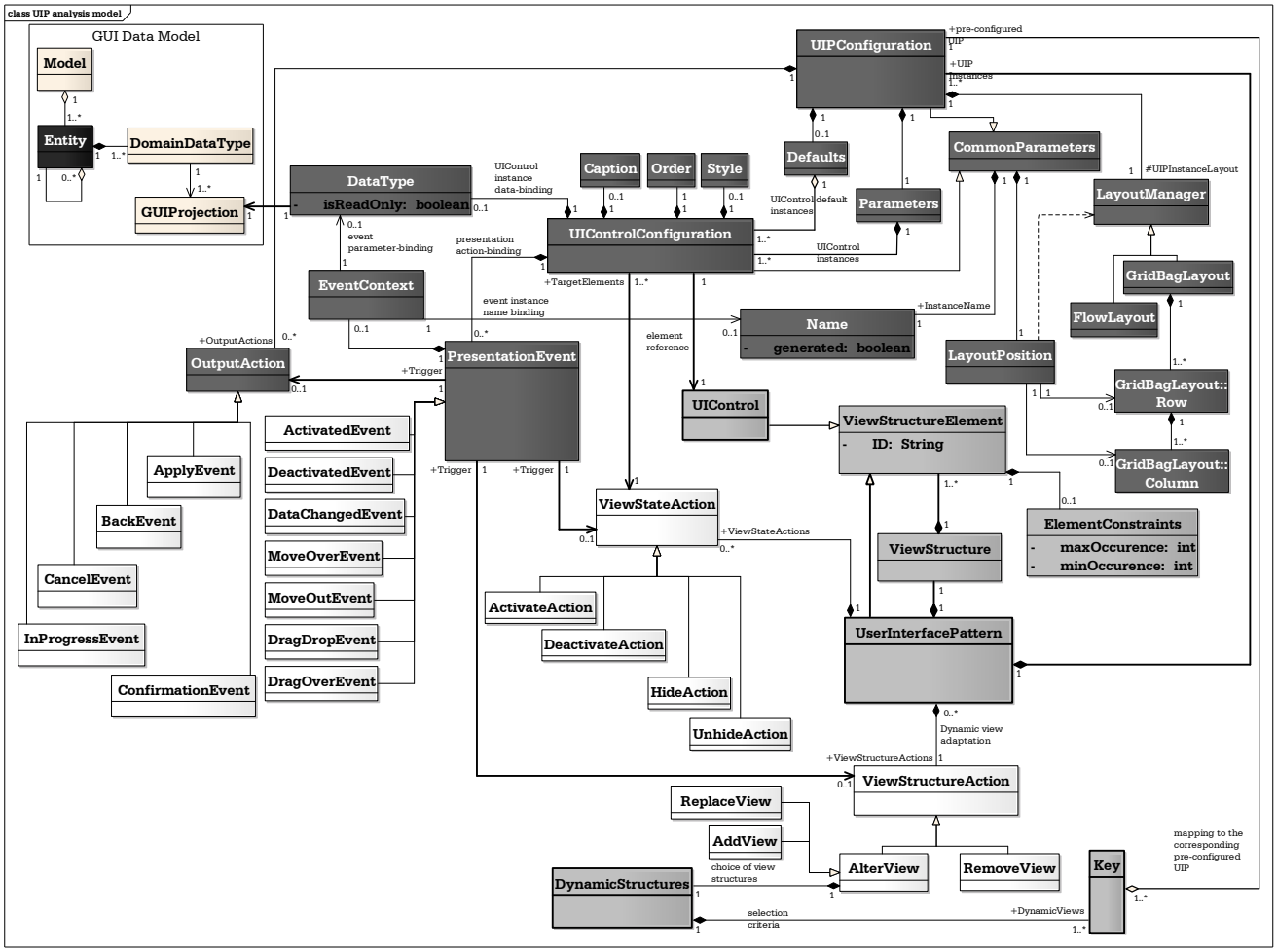

Figure 5. User interface pattern analysis model

VI. RESULTS AND DISCUSSION

Achievements. With the elaboration of our analysis model, we detailed most factor impacts of our previous work on requirements for generative UIP representations [14][4]. Accordingly, we proposed fine-grained structures, which are in closer proximity to real applicable pattern notations than pure requirements can be.

Judgment. The current state of the analysis model is quite imperfect. However, with this initial iteration we achieved a better understanding of the information needed to express UIPs and their instances. A more vivid impression on requirements, which we have modeled explicitly and are implicitly supported by current approaches employing UIPs for model-based development [4], has been gathered. Furthermore, the model already may be used to verify the capabilities of notations for generative UIPs.

The potential notation, generator tool-chain and especially the generated architecture, which may be derived in the future from the analysis model, most likely will be somewhat complex, but since they are solely intended for automated processing without manual interference, this is a trade-off for a step further to implement generative UIPs.

Again, we would like to invite other researchers to contribute either critical judgments or improvements for the presented analysis model or its requirements basis.

Unsolved control impacts. Currently, our model only supports *ViewStructures*, which consist of UIPs always being in close cooperation. Nested UIPs are not yet intended to be reused outside the specification or their super-ordinate UIP. Being aware of this barrier, we may need to define facilities such as pattern interfaces, as this was proposed by both UsiPXML [10] and Seissler et al. [12]. In this regard, the *OutputAction* may be refined to accommodate the events required for UIP inter-communication. Eventually, the *UIPConfiguration* may be supplemented by certain input types. In the end, the first three *control aspect* impacts remain unsolved for now.

Open issues. We are aware that our model needs further elaboration and especially verification. Further issues to be solved persist in the classification and delimitation of UIP specification units. The relationships among UIPs discussed by Engel, Herdin and Märtin [21] may be considered, too.

VII. CONCLUSION AND FUTURE WORK

By resuming our previous work on requirements towards a definition for generative UIPs, we drafted an analysis

model for UIPs. Together with our factor model, it may be taken into consideration for the verification of other approaches mentioned and not mentioned here. With the progression towards an improved version of our analysis model, a more general applicable model-based UIP development process may be established in the future.

Future work. For future work, we see a refining and correcting iteration for the analysis model with regard to simplicity and completeness according to all impacts. In detail, we have to assess the mandatory and optional parameters on the basis of our listed examples. Furthermore, we will concentrate on the unsolved control aspect issues.

REFERENCES

[1] X. Zhao, Y. Zou, J. Hawkins, and B. Madapusi, "A Business-Process-Driven Approach for Generating E-commerce User Interfaces," Proc. 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07), 2007, Springer LNCS 4735, pp. 256-270.

[2] S. Wendler, D. Ammon, T. Kikova, and I. Philippow, "Development of Graphical User Interfaces based on User Interface Patterns," Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 57-66.

[3] G. Meixner, "Past, Present, and Future of Model-Based User Interface Development," in i-com, 10(3), November 2011, pp. 2-11.

[4] S. Wendler, D. Ammon, I. Philippow, and D. Streitferdt "A Factor Model capturing requirements for generative User Interface Patterns," Proc. 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 13), May 27 - June 1 2013, Xpert Publishing Services, in press.

[5] M. J. Mahemoff, L. J. Johnston, "Pattern Languages for Usability: An Investigation of Alternative Approaches," Proc. 3rd Asian Pacific Computer and Human Interaction (APCHI 98), 1998, IEEE Computer Society, pp. 25-31.

[6] A. Dearden and J. Finlay, "Pattern Languages in HCI; A critical Review," Human-Computer Interaction, 21(1), pp. 49-102.

[7] J. Borchers, "A Pattern Approach to Interaction Design," Proc. Conference on Designing Interactive Systems (DIS 00), August 17-19 2000, ACM Press, pp. 369-378.

[8] D. Ammon, S. Wendler, T. Kikova, and I. Philippow, "Specification of Formalized Software Patterns for the Development of User Interfaces," Proc. 7th International Conference on Software Engineering Advances (ICSEA 12), Nov. 2012, Xpert Publishing Services, pp. 296-303.

[9] A. Wolff, P. Forbrig, A. Dittmar, and D. Reichart, "Tool Support for an Evolutionary Design Process using Patterns," Proc. Workshop: Multi-channel Adaptive Context-sensitive Systems (MAC 06), May 2006, pp. 71-80.

[10] F. Radeke and P. Forbrig, "Patterns in Task-based Modeling of User Interfaces," Proc. 6th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 07), Nov. 2007, Springer LNCS 4849, pp. 184-197.

[11] J. Engel and C. Märtin, "PaMGIS: A Framework for Pattern-Based Modeling and Generation of Interactive Systems," Proc. 13th International Conference on Human-Computer Interaction (HCII 09), July 2009, Springer LNCS 5610, pp. 826-835.

[12] M. Seissler, K. Breiner, and G. Meixner, "Towards Pattern-Driven Engineering of Run-Time Adaptive User Interfaces for Smart Production Environments," Proc. 14th International Conference on Human-Computer Interaction (HCII 11), July 2011, Springer LNCS 6761, pp. 299-308.

[13] K. Breiner, G. Meixner, D. Rombach, M. Seissler, and D. Zühlke, "Efficient Generation of Ambient Intelligent User Interfaces," Proc. 15th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 11), Sept. 2011, Springer LNCS 6884, pp. 136-145.

[14] S. Wendler, I. Philippow, "Requirements for a Definition of generative User Interface Patterns," Proc. 15th International Conference on Human-Computer Interaction (HCII 13), July 2013, in press.

[15] K. Breiner, M. Seissler, G. Meixner, P. Forbrig, A. Seffah, and K. Klöckner, "PEICS: Towards HCI Patterns into Engineering of Interactive Systems," Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 1-3.

[16] M. van Welie, G. C. van der Veer, A. Eliëns, "Patterns as Tools for User Interface Design," C. Farenc, and J. Vanderdonckt (Eds.), "Tools for Working ith Guidelines," 2000, Springer, London, pp. 313-324.

[17] J. Tidwell, "Designing Interfaces. Patterns for Effective Interaction Design," 2005, O'Reilly.

[18] E. Hennipman, E. Oppelaar, and G. Veer, "Pattern Languages as Tool for Discount Usability Engineering," Proc. 15th International Workshop Interactive Systems. Design, Specification, and Verification (DSV-IS 08), 16-18 July 2008, Springer LNCS 5136, pp. 108-120.

[19] S. Fincher, "PLML: Pattern Language Markup Language," http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html 24.03.2013

[20] 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)," Extended Abstracts of the 2003 Conference on Human Factors in Computing Systems (CHI 2003), ACM, 2003, pp. 1044-1045.

[21] J. Engel, C. Herdin, and C. Märtin, "Exploiting HCI Pattern Collections for User Interface Generation," Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 36-44.

[22] 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), June 2010, ACM, pp. 12-19.

[23] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML: An Appliance-Independent XML User Interface Language," Computer Networks, 31(11-16), Proceedings of WWW8, 17 May 1999, pp. 1695-1708.

[24] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml 12.04.2013.

[25] 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 04), 19-20 July 2004.

[26] J. Vanderdonckt, "A MDA-Compliant Environment for Developing User Interfaces of Information Systems," O. Pastor, J. F. e Cunha (Eds.): Proc. 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005), 2005, Springer LNCS 3520, pp. 16-31.

[27] F. Radeke, P. Forbrig, A. Seffah, and D. Sinnig, "PIM Tool: Support for Pattern-driven and Model-based UI development," Proc. 5th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 06), Oct. 2006, Springer LNCS 4385, pp. 82-96.

[28] M. van Welie, "A pattern library for interaction design," http://www.welie.com 27.04.2013.

[29] C. Märtin and A. Roski, "Structurally Supported Design of HCI Pattern Languages," Proc. 12th International Conference on Human-Computer Interaction (HCII 07), July 2007, Springer LNCS 4550, pp. 1159-1167.

 ISBN: 978-1-61208-276-9