The Impact of User Interface Patterns on Software Architecture Quality

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

Abstract — **Current research suggests user interface patterns (UIPs) to lessen efforts for the development and adaptation of graphical user interfaces (GUI). UIPs shall enable the reuse of both layout and interaction definitions that can be instantiated for any desired context. Most approaches are based on generative development. However, no details about target architectures or examples that prove the variability and proper structuring of UIP artifacts have been published yet. According to conventional GUI architecture development, major design decisions have to be solved individually, since no standard architectures are presently available. This applies to UIP based solutions as well, so that the target architectures are both hard to establish and maintain. On the basis of a general GUI responsibilities model, prevailing GUI design issues will be analyzed according to their impact on UIP based solutions. Furthermore, UIP specific responsibilities are identified and modeled as a software category graph. With this work, the implementation options of UIP architectures are discussed. Finally, we draft a possible solution architecture on the basis of these generalized concerns.**

Keywords — user interface patterns; model-based user interface development; HCI patterns; user interface generation; GUI software architecture; graphical user interface.

I. INTRODUCTION

A. Motivation

Domain. Nowadays, business processes build on the vast support of business information systems. These systems have to realize a large set of requirements that presume a multitude of services that are requested to handle thousands of data sets with a clearly defined stereotype structure. Depending on the domain and specialization of business processes, standard software for customizing to specific requirements or software that has to be developed individually remain as options for their IT-support.

Individual GUIs. Regardless of the chosen solution, the demand for individually designed graphical user interfaces (GUI) has to be considered as a great impact on software architecture. Proven human computer interaction (HCI) patterns [1] enable usability traits that can be essential for both user acceptance and productivity. Therefore, those patterns are to be applied to the context of dialogs, which will be coupled to the application services and data structures the users need to interact with according to business process definitions. In this context, standard software quickly is pushed to its limits concerning customization options for individual dialogs. As far as individual software is concerned, generative and model-based development has greatly advanced with respect to the creation of stereotype structures within a software architecture.

User interface patterns. However, the development and maintenance of GUI dialogs still implies high efforts. To achieve a higher efficiency on the basis of increased reuse, HCI patterns are to be formalized in order to apply them for effective generation of dialog views. On that basis, user interface patterns [2][3] (UIPs) have emerged that shall model essential HCI pattern structures. In addition, the new kind of pattern offers parameterization options in order to apply the corresponding HCI pattern to any suitable context. In sum, the application of UIPs promises many feats for future generative development

Limitations. Currently, two major issues obstruct the vast deployment of UIPs.

Primarily, the UIP concept itself has not gained sufficient maturity: the current state of formalization for UIPs is still not adequate with respect to UIP variability requirements [3], which are essential for a general application of UIPs as versatile patterns. The design of a dedicated UIP language could be initiated as an option and already was attempted [4] or is work in progress [5]. Nevertheless, high efforts are to be considered for that approach.

Besides, UIPs require a software architecture of high quality due to their high reusability and variability traits. The architecture has to be composed of a stable set of components with standardized interface structures to allow the reuse of UIPs within and among different software projects. Thus, UIPs need to be integrated into an encapsulated structure within the GUI sub-system, so that the realization of workflows, functional requirements and related application components is not affected. Ultimately, UIPs have to be decoupled from their application context. The current research into GUI architectures does not provide such an architecture and approaches that are already based on UIPs have not published details of target architectures yet. We will briefly reason about that architecture concerns.

Architecture concerns. Available patterns [6] and related sources [7][8] provide valuable aspects for design decisions, but they are rather isolated and have to be integrated into one comprehensive reference architecture that allows the seamless integration of UIPs. In this respect, common MVC variants and the Quasar client reference architecture [9] are too general in concept [10], so that major design decisions are still to be elaborated in order to allow the effective deployment of UIPs.

Moreover, the technical GUI frameworks already define some architecture constraints for action- and data-binding, as well as control facilities. So, the architect has to find ways to limit their influence on the variability of UIPs, otherwise

 ISBN: 978-1-61208-367-4

UIPs would only be applicable in a certain technical environment.

Ultimately, the development of a high quality software architecture on the basis of a clearly defined requirements structure takes considerable time and has to mature by the experience gained during several projects. Often budgets are just as high to barely exceed break-even or reuse is not envisioned or planned [11], and so, hardly any efforts remain to build and refine reference architectures in the aftermath. In the end, this tasks remains for academic research.

B. Objectives

With our previous work on UIPs [12][3][13] and general GUI architecture responsibilities [10], we have a solid foundation to approach the above introduced UIP- and GUI-architecture limitations.

Firstly, we have to consider that UIP based solutions heavily rely on a pre-defined architecture to accommodate code structures build from both the pattern and instance or configuration information. Consequently, we have to analyze the major GUI design decisions and identify additional responsibilities required for the implementation of UIPs.

Since model-based approaches are already work in progress, we will have to critically discuss the principal UIP implementation architectures. Accordingly, we will criticize the general formalization approach and argue for an alternative solution. As a consequence, we draft a suitable GUI reference architecture based on the new UIP concerns.

C. Structure of the Paper

In the following section, related work that is relevant for our objectives is presented. The third section presents our analysis of the impacts UIP based solutions have on the general GUI design issues. In addition, a software category model is described that details the UIP specific responsibilities of a GUI architecture. In Section IV, the principal UIP implementation alternatives are discussed. A UIP based architecture is drafted in Section V, before we present our results in Section VI. Finally, we draw our conclusions and state future work in Section VII.

II. Related Work

A. Standard GUI Architecture for Business Information Systems

Siedersleben and Denert [14] already tended to the missing GUI architecture standardization issue outlined in the introduction. To enable a more effective design with respect to separation of concerns and increased adaptability to changes, business information systems had to be designed on the basis of a standard architecture, which would incorporate a defined set of patterns and interfaces.

One of those patterns of the envisioned standard architecture was the Virtual User Interface (VUI) that is depicted in Figure 1. The VUI should allow a developer to implement dialogs with a high independence from the rendering *GUI Framework*. In detail, a *Dialog* and its events should be implemented with the aid of the technical independent, abstract interfaces *WidgetBuilder* and *EventListener* rather than using certain interfaces and objects of the imported *GUI Framework* directly. The primary goal was to preserve the interchangeability of the *GUI Framework* without affecting existing dialog implementations. Solely the component *Virtual User Interface* would interact directly with the *GUI Framework*, and thus, would depend on technological aspects.

The basic concepts worked as follows. A *Dialog* would create and even adapt its view at runtime with the operations provided by *WidgetBuilder*. The *VUI* could be delegated by the *Dialog* in order to construct and configure a new status and button bar inside a specified frame. Moreover, the *VUI* would notify the *Dialog* via the interface *EventListener* when events would have been induced by UI-Controls. More details are not known.

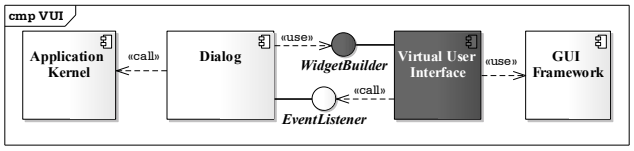

Figure 1. Virtual user interface architecture as introduced in [14].

B. GUI Software Categories and Design Issues

No further ideas for the standardization of an architecture for the domain have been published. A GUI reference architecture [15][9] (Quasar client) and a concept for the identification of components as well as their interface design [15] were presented instead. The latter was based on software categories that would mark the responsibilities and dependencies of a given component. These categories could be used to valuate the cohesion of a given modular structure according to the separation of concerns principle of design.

In [10], we applied the software category concept for the identification and delimitation of general GUI responsibilities. In this regard, the common MVC variants [16][17] and the Quasar client architecture [9][15] were considered both for analysis, and besides other sources, the derivation of software categories. The resulting software category hierarchy and their dependencies are illustrated in Figure 2. The related sources mostly separate the very basic categories *0* (a programming languages' reusable foundations), *A* (application, domain) and *T* (technical aspects, frameworks) without any refinement. Being based on these general software categories, each refined software category of Figure 2 represents the knowledge required for implementing the operations, their proper sequence and required data structures for the respective responsibility they are entitled with.

As a result of our analysis, we derived three major GUI design issues. Firstly, the architect has to decide on how much application control flow is assigned to GUI dialogs and how they coordinate the interaction with the application kernel. This would also influence the application related event processing, and in particular, the update of presentation view states due to changed application data (another view for a certain use case step). Secondly, for the navigation among dialogs and flow of the sub-dialogs a dedicated controlling component has to be allocated. Thirdly, the transformation of application aspects like data models and the visual representation of domain model entities have to be solved. In this regard, a tight coupling to technical frameworks should be limited. For details, [10] can be consulted.

 ISBN: 978-1-61208-367-4

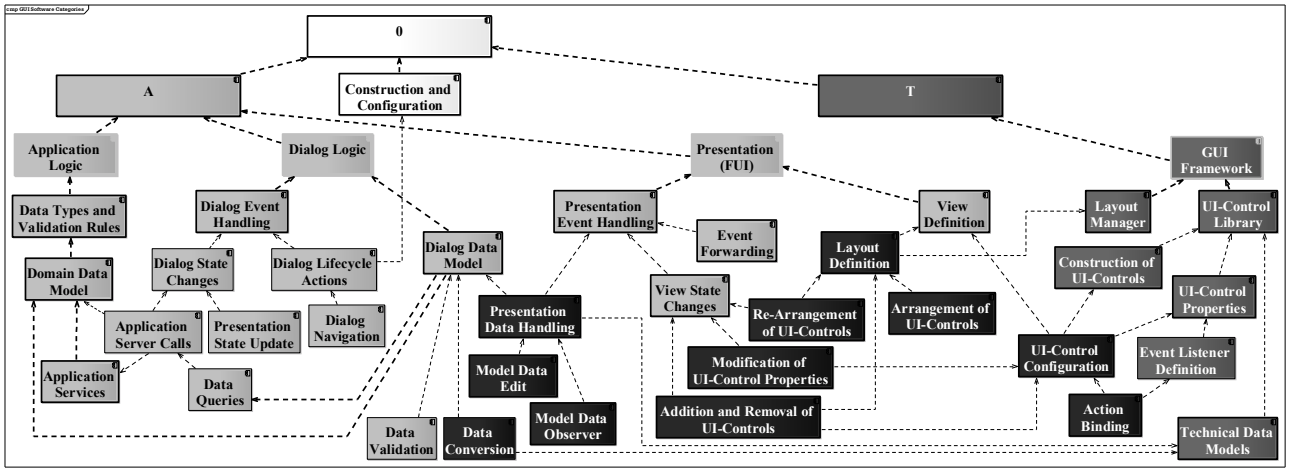

Figure 2. General GUI responsibilities modeled via software categories.

C. User Interface Pattern Aspects

Past work in the field of HCI resulted in the combination of the specification of reusable GUI visual design and interaction solutions with a pattern-based description. Several pattern languages emerged [18]. Current research is trying to exploit these patterns for the automated generation of GUIs. As a consequence, UIPs are based on the idea to formalize HCI visual designs into software patterns that can be reused in any desired application context.

In [12], we elaborated the theoretical and practical implications of that kind of pattern applied within the general transformations from domain requirements to a final user interface specification. As result, UIPs are very promising for bridging the gap between pure requirements and potential GUI specifications, since they define many aspects like used UI-Controls and their interaction designs. Particularly, the latter can be reused to imagine and prototype GUIs of high usability. Moreover, we presented and discussed general architectures for the practical application of UIPs.

With our contributions [2][19], first criteria and aspects for the UIPs to be deployed for variable GUI dialog generation were introduced. Based thereupon, we formed a drafted definition of that particular artifact. The UIP requirements were considerably refined in [3] by the description of an influence factor model. Particularly, the abilities of UIPs were defined by the three aspects *view* (visual elements, layout), *interaction* (view states, events and data-binding) and *control* (composition and interaction of UIPs, binding to application relevant events).

Lastly, the UIPs aspects were further detailed by an analysis model [13], which was derived from the impacts of the influence factor model and describes the resulting structure of a UIP. The elaborated structure could be positively evaluated with UIP examples illustrated by object models. With that last step, basic foundations of UIPs and many aspects that are essential for the formal expression of a UIP are available now.

D. Model-based Frameworks on the Basis of UIPs

Past research has put considerable efforts into the deployment of UIPs or closely related patterns within model-based developments processes.

The University of Rostock [20] mainly worked on the derivation of dialogs from task models and included UIP-like artifacts called PICs (pattern instance components) for the generation of final views. A dedicated UIP formalization language on the basis of UsiXML [21] called UsiPXML [4] was created in parallel. A continuation is not known.

The University of Augsburg presented research into UIPs with the introduction of an own modeling framework called PaMGIS [5]. To express UIPs, a dedicated DTD was partly presented in [5]. The work is still in progress.

The University of Kaiserslautern focused on the application of UIPs for the domain of production environments [22] and sought a way of enabling GUI devices to be able to adapt their view at runtime [23]. In their approach, UIML [24] as a basic GUI specification language is used and augmented with a pattern interface and configuration facilities to be interpreted at runtime. There are only few details of the modeling framework [25] published.

In sum, all approaches suggest individual modeling frameworks that rely on specific formalization formats of UIPs and produce different outputs. A detailed review of these approaches compared to our UIP requirements model is provided in [3].

III. User Interface Pattern Architecture Impacts and Responsibilities

A. Approach and GUI Software Categories

Due to the prevailing issues in GUI architecture design, the development of a new reference architecture for UIPs is most likely to be approached. The interfaces between components need to be harmonized to fit UIPs as reusable entities that may be exchanged to allow the quick adaptation of GUI dialogs. In addition, the event processing has to be prepared to allow the exchange and re-configuration of UIP instances. Finally, UIPs will require a new quality of the software architecture with additional responsibilities.

Category refinement. For the design of such a reference architecture, it is of the essence to consider the separation of concerns. To prepare a proper component identification in this context, the software categories presented in Section II.B will be of great value. They already incorporate the basic

separation of application and technology as requested in [9][11][14][15] as far as possible. In addition, they feature fine-grained refinements of both areas of knowledge. This is essential to avoid coarse grained software categories that concentrate too many responsibilities and would not improve traceability. With coarse grained software categories the component identification would not guarantee separated concerns, since components eventually would have to be refined on the fly during implementation. So, the creation of traceability-links would rely on the coarse grained architecture models, and most likely, would not result in a detailed impact analysis. In contrast, the categories of Figure 2 were separated to a more fine grained level that is able to guide the decisions for GUI design issues. Especially the event processing was differentiated concerning the context (*Presentation*, *DialogLogic*), triggers (*Dialog Event Handling* and children) and execution (*Application Server Calls*, *Dialog Navigation*, *View State Changes*) of state changes. Furthermore, our analysis of the Quasar client with the aid of the software categories in [10] revealed several open issues that were due to lacking details or cohesion.

B. The Impact of User Interface Patterns on GUI Design Issues

We discuss how UIPs will impact the GUI design decisions and ultimately affect the identified responsibilities.

A-T-separation. Foremost, UIPs will stress the separation of *A* and *T* categories due to their variability: If UIPs are bound to a certain *GUI Framework*, they will be virtually rendered useless for architectures employing other technical environments.

Besides this very basic separation, an additional separation has to be considered between *Dialog Logic* and *Presentation* design. To allow the quick adaptation of dialogs, the logical part of a dialog (dialog kernel) has to be able to interact with a presentation that may be altered in design frequently. The former should not be affected when the presentation design was changed to an alternative set of UIP instances. For instance, two large panels for editing data in a single dialog were re-structured into a dialog featuring two tabs instead. Thus, both logic and visual dialog parts have to be decoupled for the adaptation of UIPs.

Flow of application logic. Concerning the division of labor between application and dialog kernel, UIPs need a single basis for coupling of their generic events to context specific behavior. The *OutputActions* of a UIP [13] should be processed centralized by a single component like the dialog kernel to preserve the exchangeability of UIPs emitting those events from the variable presentation part. In this regard, the category *Presentation State Update* gains importance and shall enable a dialog kernel to govern visual changes regardless of the concrete *Presentation* implementation and its UIP instances. The further rationale is to decouple application-independent UIP events from application specific interpretation and processing. In principle, a UIP may be configured to emit an event that may be interpreted very differently in various dialog kernel contexts. With respect to UIP combinations that form one *Presentation* in interaction with the dialog kernel, the individual UIPs have to be kept independent from each other to allow for flexible combinations. One UIP shall not limit the flexibility and change of states of another. In return, a UIP needs a standardized interface to application related artifacts for *Event Forwarding*.

Besides event handling, this also applies for the *Data-binding* impact [3] UIPs require. Obviously, the dialog kernel will become a direct interaction partner for both events and data of a number of UIP instances that are to be integrated together instead of a single view or *Presentation* unit. Therefore, the context for UIPs has to be kept rather isolated from application kernel components, what allows versatile combinations between both. Finally, it has to be considered to centralize the flow of interaction specified by use case models [26] in order to keep an implicit but recognizable connection between UIPs and those functional requirements. In this regard, the dialog kernel may serve once again as central unit that coordinates both *Application Server Calls* and *Presentation State Updates*. The latter establish the implicit connection between UIPs, their states or instantiation and use case steps.

Navigation. The scope of UIPs can be limited to visual elements within dialogs or can even span entire dialog types and their navigation. The different UIP abstractions are symbolized by the various pattern types defined in model-based frameworks [5][23][27]. For the implementation of UIPs that trigger and design dialog navigation like wizards or tabs [28], a dedicated component will be needed that translates the events emitted from these UIPs into the desired change of views or dialogs. The rationale for the centralization is that UIP instance combinations can be very versatile, though UIPs only define the UI-Controls that can be assigned to trigger navigation events. Finally, the evaluation of these events has to be governed by the same component that implements the navigation for non-UIP dialogs in order to allow the seamless integration of UIPs with ordinary dialogs. According to the software category tree, the respective responsibility belongs to the task set of *Dialog Event Handling*, since the navigation is restricted by validation results. For instance, each wizard dialog needs valid inputs to allow the navigation to the next step.

UI-Control set. A further aspect raised by UIPs is the availability of certain UI-Control implementations. For every domain or project, a range of certain UIPs is of relevance. These are to be defined on the concrete user interface (CUI) level of abstraction [12] with reference to [29]. Therefore, the UIPs have to be transformed into UI-Control compositions on the final user interface (FUI) level [29] of abstraction. The CUI based implementation of UIPs ensures their platform independent application and decouples them from *GUI Framework* specific concepts. However, UIP basic elements must be covered by the favored GUI frameworks. For instance, one cannot expect to develop UIPs on the basis of Java AWT due to the very limited set of UI-Controls.

 ISBN: 978-1-61208-367-4

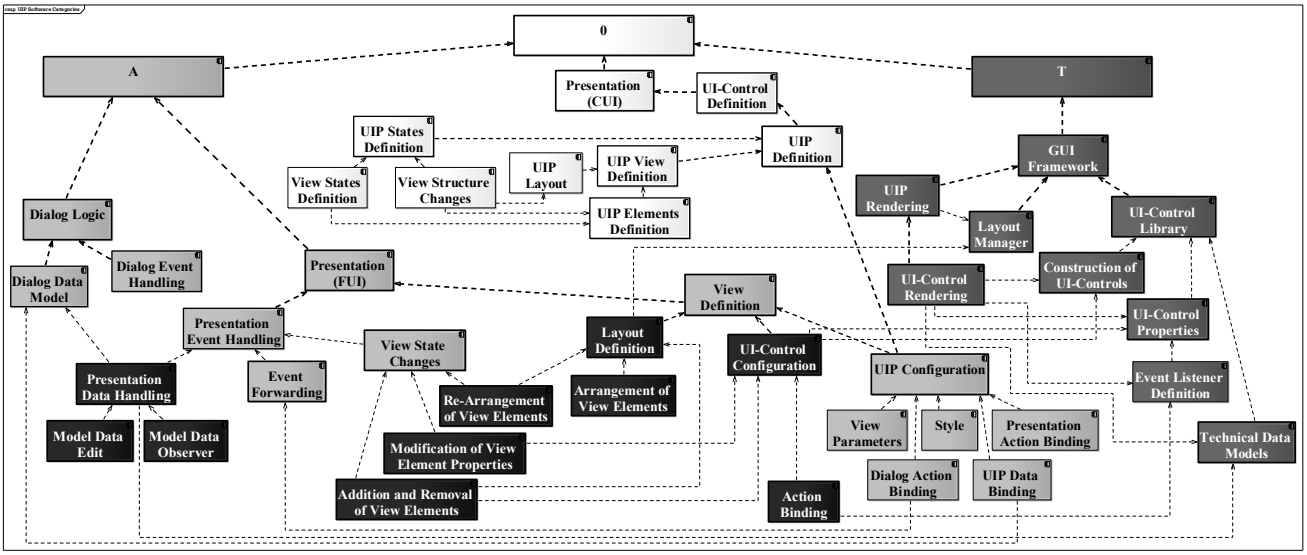

Figure 3. GUI software categories enhanced with UIP based responsibilities.

C. User Interface Pattern Responsibilities Model

From the basic foundations of our previous work and the above mentioned design aspects, the UIP related responsibilities of a GUI architecture will be developed.

Factor model. The influence factor model for UIPs [3] describes additional requirements besides the general GUI architecture responsibilities. As a consequence, the software categories have to be enhanced to reflect the configuration and variability aspects of UIPs. The resulting software category tree is essential for the identification of components of the UIP implementation, the planning of their dependencies and the consideration of UIP requirements. Finally, the category model will translate the factor impacts to comprehensive categories of software component design.

Analysis model. The UIP analysis model [13] represents detailed structures that refine the impacts of the influence factor model. In detail, the analysis model describes coupling points between GUI architecture and UIP configuration facilities, basic structures for UIP units and detailed parameters for visual and behavior aspects. According to the software category identification, the information is useful to mark dependencies to existing basic GUI responsibilities. Afterwards, the analysis model will drive the design of the final UIP representations rather than the software categories.

A-T-separation. The enhanced software category tree is depicted in Figure 3. It is apparent that the UIP software category tree is largely influenced by the mandatory A-T-separation impact. This results into a new hierarchy of *0* software categories. The *Presentation (CUI)* defines the view elements to be reusable in any project. In detail, the *UI-Control Definition* is essential to provide a generally available set of UI-Controls as building blocks for the definition of UIP units. Therefore, *UIP Definition* is dependent on the former. The other categories that refine *UIP Definition* are directly derived from the impacts of the influence factor model. In general, the *0* based categories only define the reusable elements, their properties and abstract behavior, but no final user interface is implemented.

Furthermore, the new *0* category elements can be declared to be used for the *Presentation (FUI)* via *UIP Configuration*, but the rendering has to be implemented for the chosen platform individually. Therefore, the *T* software categories *UI-Control* and *UIP Rendering* were added. These depend on *GUI Framework* sub-categories like this is the case for the conventional *Presentation (FUI)* categories [10].

The ordinary *Presentation (FUI)* composition usually consists of four basic operations: The construction of new UI-Controls and the setting of their properties (*UI-Control Configuration*), the addition of the new UI-Control to a superior container like a panel or frame (*Arrangement of View Elements*) and the optional definition of an event listener (*Action Binding*). All these operations are bundled into respective *AT* software categories, which directly combine domain specific knowledge (content, properties and placement) with technical operations (construction, auxiliary objects like layout constraints or scroll panes) later in code.

When UIPs are instantiated, the above basic operations are distributed among reusable pattern information (*UIP Definition, UI-Control Definition*), context specific configuration (*UIP Configuration*) and the technical rendering (*UIP* and *UI-Control Rendering*). The *Renderings* do not depend on the respective *Definitions*, since they are solely in charge of either the construction of new UI-Controls (*UI-Control Rendering*) or the arrangement of a specific layout (*UIP Rendering*). For that purpose, the *Definitions* define and use basic parameterized operations for their content that are finally implemented by the respective *Renderings*. The *Definitions* contain operations of higher order and the *Renderings* consist of rather atomic ones, hence a *Definition* command will be translated by the *Renderings* into multiple *GUI Framework* calls. Thus, the technical details that are usually present in the ordinary *View Definition* sub-categories are encapsulated by the *Renderings*. The *UI-Control Rendering* will be called with complete information based on the UIP instance parameters, so that only complete units can be created with the *Definition* commands. In this context, the *UIP Configuration* gathers

 ISBN: 978-1-61208-367-4

parameter data with knowledge about the *UIP Definition* (what parameters are exactly present). After the parameters have been configured, they are passed to the *UIP Definition*, which contains all necessary commands in proper order for UIP instantiation and finally delegates the *Renderings*. The latter will implement the abstract operations of the *UIP* or *UI-Control Definition*. The *UIP* and *UI-Control Rendering* are in analogy to *View Definition* sub-categories *UI-Control Configuration* and *Layout Definition*: UI-Controls do not define the gross layout. This instead is a task of the higher situated category *Layout Definition*. Accordingly, there is the distinction between both *Renderings*.

This order of operations is not obvious from the software category graph, since this kind of modeling lacks a runtime or sequence view. In this regard, the dependencies of Figure 3 do not describe calling sequences. The dynamic aspects of calls and the purity of categories will be better visible with component diagrams that describe interfaces and feature the assignment of categories to components. Eventually, the component and interface modeling will refine and verify the software category model.

Changes were applied to sub-categories of *View Definition* and *View State Changes* of *Presentation (FUI)* to reflect the widened set of available view elements. The responsibilities now apply both for UI-Controls and UIP instances.

Flow of application logic. Concerning the flow of application logic and the integration of UIPs, the *Dialog Action Binding* is dependent on *Event Forwarding*, since the ordinary facilities of the event processing chain [10] have to be reused by UIPs and shall not be influenced by a conflicting solution. With respect to the Quasar Client reference architecture [15], the dialog kernel is likely to receive UIP application events in parallel to events from ordinary *Presentations*. In principle, for any UI-Controls of a UIP *PresentationEvents* can be defined [13]. During configuration of UIP instances, application relevant *OutputActions* can be assigned to these events [13]. To preserve this variable binding of UIPs and their events to application behavior (*Dialog Logic*), UIPs have to be decoupled from application logic. This is achieved by the following concepts. Initially, *PresentationEvents* have to be configured for UI-Control instances to be deployed within a particular UIP instance. These can be used to model a trigger for either *ViewStateActions* or *ViewStructureActions* that may add or remove view elements during runtime [13]. In addition, particular *PresentationEvents* can be linked to *OutputActions* that are relevant for application logic (*Dialog Event Handling*) outside the UIP instance. A further decoupling is achieved by the separation of *Event Forwarding* (notification of an event), the decision of a proper reaction by *Dialog Event Handling*, and finally, the implementation of resulting state changes, e.g., *View State Changes* of the *Presentation* [10]. In other words, two states of knowledge are separated: Firstly, what and when events are to be reported. Here, the *OutputActions* mark those events of relevance. Secondly, how will be the reaction implemented that corresponds to reported events. Ultimately, this separation of concerns will allow either the integration of UIPs or ordinary *Presentations* as sender of events relevant for application behavior. This design will allow the versatile configuration of UIPs and their exchangeability. However, a dedicated receiver is essential, which processes events and interacts with application components.

Navigation. In analogy, the navigation design has to follow the same concept: a UIP may emit events that are translated into resulting navigation by a dedicated component. Both concepts preserve the later exchangeability of UIP instances, and thus, allow the decoupling of *Presentation (FUI)* and *Dialog Logic*.

Summary. Finally, UIPs require a GUI architecture that provides a working infrastructure for *Application Server Calls*, *Dialog Navigation*, platform-specific implementations of their UI-Controls and facilities for event as well as data binding. In fact, UIPs can only be applied to describe certain configurations. Thus, the situational meaning of this information is out of the scope of reusable *UIP Definitions* but is to be processed by existing GUI components based on common responsibilities like those modeled in Figure 2. Accordingly, UIP solutions will be based on many common GUI software categories. Therefore, the basic GUI design decisions presented in [10] and discussed here for UIPs in the previous section have to be solved prior to any UIP implementation. Ultimately, UIPs need an elaborate GUI reference architecture with a clearly defined component structure as suggested by the software category model of Figure 2: the new responsibilities are merely enhancements with many dependencies to the basic categories. Particularly, the differentiated categories for event processing [10] will be an essential basis for flexible UIP integration.

The categories partly may be too fine grained, but these serve their purpose better than coarse grained ones that lead to less cohesion and less effective tracing. In contrast, the fine grained categories may later serve as units for lower level design like classes or even operations.

Anyway, the control aspects of UIPs [3] are not modeled here besides *Dialog Action Binding*. This is due to these aspects are cross-cutting concerns that need further elaboration on the basis of detailed examples.

D. Virtual User Interface reviewed

To solve the A-T-separation and maintain the purity of software categories, the virtual user interface from Section II.A is considered.

The main idea of Siedersleben and Denert [14] was to abstract common operations needed for the communication with technical GUI components into lean and easy to reuse interfaces that would considerable simplify the usage of complex APIs or associated frameworks. This concept could yield several benefits when applied for UIP instantiation.

Firstly, the VUI allows the implementation of styleguide rules [14] and other related layout specifics. Therefore, the created layout corresponds to specified rules and could be augmented by standard presentation elements like status or button bars whenever UI-Controls or entire dialogs are requested to be build. This scope of pre-defined GUI layout and selection of UI-Controls can be extended to enable the creation of UIPs. For given UIPs, common UI-Control elements or even nested UIPs that occur regularly as children can be realized as ready to reuse compositions as well.

 ISBN: 978-1-61208-367-4

Secondly, the VUI is worth a consideration for UIPs, since its suggested way of dialog implementation conforms to the concrete user interface model level (CUI) of the Cameleon reference model [29]. This level of GUI modeling foresees certain types of UI-Controls, which may be a common intersection of the ones that are offered by several popular GUI frameworks. Besides, these UI-Controls remain independent from a platform specific implementation as this is the main emphasis of the VUI. Ideally, available UIP implementations could be reused together with alternative GUI-Frameworks.

Thirdly, when the main idea behind the VUI and its interface operations are fully complied with, both basic UI-Control creation and UIP instantiation will have to be realized resulting in a hierarchy of GUI building operations. Therefore, the basic VUI interfaces are relevant for the bottom-up composition of UIPs. Additionally, non-UIP based dialogs could be created at the same time.

However, no details and implementations have been published for the VUI yet. It remains as a general pattern only and solutions must be drafted individually. In particular, the involved interfaces have to be standardized for a GUI system and its dialog types. This step is of the essence, since it permits the reuse of reoccurring functionality such as the creation of views with common UI-Controls and their binding to events. To conclude, the essential elements the GUI system presentation component will constitute of have to be abstracted very clearly and completely in order to provide a CUI level model suitable for the domain.

IV. Discussion of User Interface Pattern Implementation Options

A. Criteria

The principal architecture concepts for UIP implementation were briefly outlined in [12]. Accordingly, we distinct the two concepts of model-based generation and a solution being based on the virtual user interface architecture described in Sections II.A and III.D. We will discuss these alternatives in the light of the GUI design issues and more recent state of the art. The criteria to be considered are presented below.

The primary criterion is the UIP formalization and its completeness. All structural properties and variability aspects of these patterns [13] should be expressed by the chosen notation. Finally, UIPs should be expressed by a CUI model to preserve the platform independent specification.

The second criterion considers the target architecture and respective assumptions. In detail, the integration of UIP instances with other architecture artifacts, which affects the major GUI design issues, is reflected. Since UIPs mostly assume presentation responsibilities, their interface to application logic has to be lean to ensure a variable presentation without affecting application components. For the sake of adaptability, the *Dialog Logic* and associated navigation control should be decoupled from specific UIP instances, too. To preserve the option to integrate non UIP-based dialogs, the decoupling is essential.

A third criterion considers the required tools, and lastly, the coupling to a certain platform and potential reuse of concepts are considered.

B. Model-based Generation

Formalization of UIPs. The model-based frameworks introduced in Section II.D employ their specific format for expressing UIPs for the generation of GUIs. It is noteworthy that the capabilities of the applied notations are not published completely or mentioned at all. In addition, no detailed examples that proof the variability, composition ability and reuse of formalized UIPs have been published yet. Therefore, the maturity of the generation based UIP approaches surveyed in [3][13] was valuated as insufficient. The model-based generative frameworks still seem to be challenged by the full expression of all required UIP aspects and are obliged to deliver a proof of concept by the evaluation of a set of representative UIPs.

Target architecture. Currently, there are no details available of the assumed architecture and integration of UIPs therein for the model-based generation. The task modeling and derivation of dialog structures often is focused by examples. In this regard, we wonder how complex *Dialog Logic* can be implemented, which demands for a number of branches due to user choices and results in different navigation options among UIP instances. Thus, it is not certain how closely task models and chosen UIPs for presentation of dialogs are coupled. In general, the complete configuration process of all related artifacts (tasks, dialogs, UIPs, application data and services) for the realization of a use case remains unknown. Lastly, it is uncertain whether manually implemented dialogs can be integrated among generated code or if every dialog specification results in the mandatory formalization of UIPs that may be used only once.

Tools. The generator based solutions require vast tool support for formalization, configuration or instantiation and finally transformations of UIPs. For the latter, two steps are necessary as UIPs and their parameters have to be transformed to a CUI model first, which is later used for final code generation or interpretation. There will be high efforts for maintaining the tool chain as well as related overhead for the definition of metamodels, rules and syntax validation. To integrate non UIP-based dialogs the developers will have to provide additional CUI specification facilities.

Platform. By using platform-independent models, the coupling of generation based solutions to certain infrastructures is generally low. Mostly, the paradigm of the GUI may be fixed to WIMP [2]. Thus, the UIP formalization is highly reusable. However, for each target platform suitable architectures and code templates have to be developed. Most parts of the generator code will be platform-specific transformations that are unlikely to be reused.

C. Virtual User Interface

Formalization of UIPs. In contrast to the generative approach, the VUI based solution does not necessarily depend on a separate notation for formalization. The formalization is realized by object-oriented CUI level code of the target platform programming language instead. We are inclined that an object-oriented language offers strong concepts that permit the vast flexibility of UIP expression. For instance, abstract classes with partly implemented operations may serve as ideal templates for UIP definitions.

 ISBN: 978-1-61208-367-4

The parameters for context adaptation can be set by operation parameters or separate setter operations. Furthermore, both structural and behavioral aspects can be combined in one specification unit. These basic facilities would have to be re-created by an external notation for a generation based solution. In this regard, even the template offering UIML 4.0 [24] GUI specification language lacks sufficient parameterization for UIPs [12] and would have to be extended.

In contrast, an OO programming language offers elementary functions to express any purpose or structure that may be improved by architectural or design patterns. Furthermore, the usage of an OO language for UIP expression is comparable to directly programming with a certain GUI framework to fulfill a certain domain's GUI requirements. Similar elementary facilities can be incorporated with the identified software categories for UIP expression, so that a high flexibility is achieved. The basic operations for presentation definition are based on the CUI level [29] and represent abstractions of common GUI framework facilities. They will both enable an accurate and abstract UIP formalization with a high flexibility due to the full range of OO language capabilities.

Initially, the UIP expression can be probed on the basis of the UIP analysis model [13]. The conceptual UIP modeling can be improved gradually without the need to adapt a specific notation and associated tools. With the basic foundations of factor [3] and analysis model [13], a rich information basis for UIPs is available that can be successively translated to code with the aid of the software category tree of Figure 3.

Target architecture. The VUI architecture is limited to presentation related tasks and does not include any assumptions concerning application integration. That means, each GUI design issue has to be solved from scratch or by the adaptation of available reference architectures. A solution tailored for UIP integration induces additional efforts but may result in an appropriate and reusable architecture.

Tools. The VUI needs no tools at all besides a compiler and an IDE that partly does the checking of programming language syntax. For visual impressions of defined UIPs, default configurations can be implemented, which may be used as test cases, too. The testing of UIP instances does not require additional inputs from external tools. The combination of UIPs and ordinary dialogs is possible without further adaptations.

Platform. For the VUI solution, the target platform language is fixed. There may be additional frameworks required, which permit the integration with different languages or even paradigms. But with each change of target language or GUI frameworks, the specific code for rendering has to be re-implemented. Therefore, the UIP formalization appears to be less reusable like the format used for the generation-based approach. But it may be ported to OO languages with comparable facilities, since the architecture is the key reusable artifact. In this regard, the architecture is based on interfaces and object-orientation, so that the VUI CUI components may partly be ported among different OO languages. Moreover, the formalization of UIPs is solely based on architecture components, interfaces and their interaction, so that no notation has to be adapted. In the end, the VUI solution may promise more reusable concepts, since they are not platform-specific like the transformations of a generator basis.

D. Outlook

The model-based generation approach raises many open issues concerning the UIP formalization and target architecture details. It is not certain when and what solutions are to appear. So, we opt for an alternative solution that is based on the VUI architecture.

V. Virtual User Interface Architecture Draft

As a result of the positive appraisal of the virtual user interface architecture, we will elaborate an architecture draft in the following paragraphs. The primary basis for the identification of components and their dependencies are provided by the software category models of Figure 2 and Figure 3. These categories need to be assigned to new components and their interfaces. The latter will clarify the dynamic behavior, which was not obviously described by the category trees. For the sake of keeping reference to the category trees, a similar naming of components was applied. In Figure 4, the structural architecture model is presented. Please note that not every software category will be represented as a component. The granularity of categories differs, so that some are assigned to components, classes (not visible here) or a set of operations modeled by interfaces.

A main component is modeled by the *Dialog*, which initiates application related behavior (*Dialog Logic*) and handles domain data (*Dialog Data Model*). Concerning the configuration of instances and initialization of visual components, the *Dialog Lifecycle Actions* are in charge.

Another main component is embodied by the *Presentation (FUI)* that serves as the final user interface with visual appearance and respective event handling. There exist two options for the instantiation of visual elements: Either simple UI-Controls can be initialized by the *UI-Control Configuration* or UIPs can be configured by *UIP Configuration*. Both components are associated to *Presentation Event Handling* to be able to have their elements linked to event processing. Triggers and state changes are decoupled by the separation of *Presentation Event Handling* and *View Definition*. The interfaces called by *View State Changes* represent operations that implement the results of visual state changes. When the received event is out of scope of the *Presentation (FUI)*, the *Event Forwarding* will call *Dialog Event Handling*. Moreover, the *Presentation Data Handling* is realized by the observer [31].

The *Virtual User Interface* component consists of one reusable (*Presentation (CUI)*) and one technical dependent (*Rendering*) component. As a consequence, there are always two representations of one UIP or UI-Control. The CUI level components of the *Presentation (CUI)* define the logical part of instances. In contrast, the *Rendering* creates corresponding technical instances that depend on the current *GUI Framework*. To decouple the CUI components from technical aspects, the *UIP Elements Definition* and *UIP Rendering* interfaces define the atomic operations required for both *UI-Control Definition* and *UIP Definition*.

 ISBN: 978-1-61208-367-4

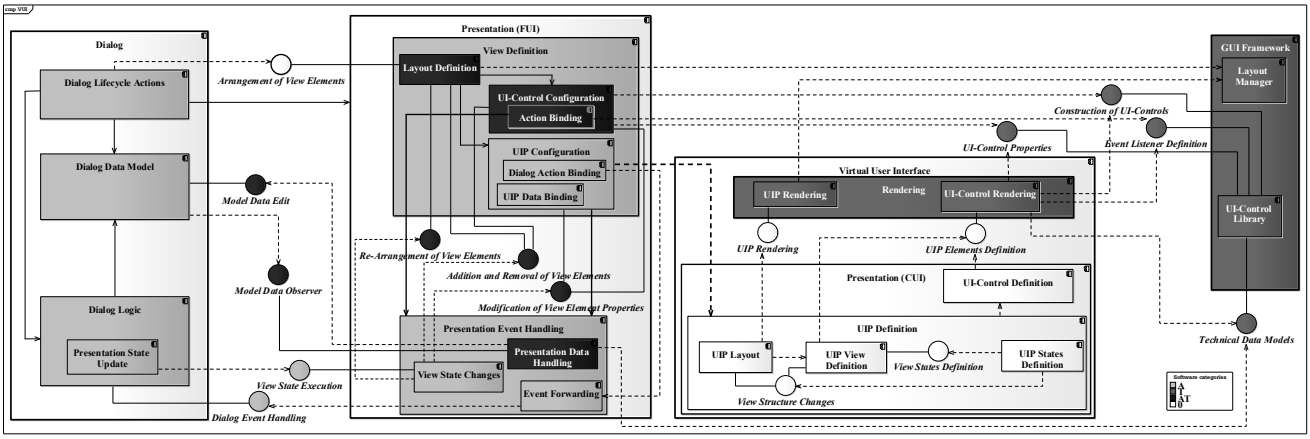

Figure 4. Virtual user interface architecture based on the UIP software categories.

These may be implemented by different *Rendering* components, which are specific for a certain *GUI Framework*. The versatile UIP formalization options are mostly assigned to *UIP Layout* and *UIP View Definition*. Depending on the current UIP instance configuration *UIP States Definition* may call the former components to trigger changes in visual or structural state.

VI. Results and Discussion

The vision to employ UIPs as reusable assets for a reduction of GUI implementation efforts cannot be realized by recent approaches because of the limited formalization of UIP aspects and variability. Besides, the general GUI design issues presented in Section II.B still persist due to the lack of detailed reference architectures and standardization. In Section III.B, we clearly pointed out how these issues impact the architecture for seamless UIP integration. A tight coupling to GUI frameworks can limit the UIP applicability. Also, important architecture concerns UIPs are connected with are without standardized solutions: navigation and application logic flow. Eventually, the integration of UIPs into GUI architectures has to overcome these issues.

Since UIP based solutions largely depend on reuse of basic GUI architecture concepts, UIP specific concerns have to be integrated and separated to reduce dependencies. In this context, we presented an enhanced software category model that addresses the prevailing GUI design issues and models typical UIP responsibilities. These categories can be used to identify a component based architecture for UIP implementation with separated concerns and limited dependencies. The identified *0* categories can be either generative sources or CUI level code of a VUI. In the end, the UIP category tree can also be helpful for generative development as it may identify aspects or components and separate them in order to enable a better maintenance of generator architectures.

As result of our comparison of general UIP implementation approaches, we opted for the unique VUI solution. The VUI solution promises a high flexibility of UIP formalization, platform independence and no additional tools or notation development efforts. On that basis, simple and complex UIPs can be relatively quickly probed for implementation. Please note that our analysis of mature XML GUI specification languages [12] revealed major limitations concerning UIP formalization that are hard to solve. UIP definitions may be better approached with OO language code.

Our VUI draft left the impression that much CUI based abstraction of common GUI framework concerns is required and that a complex architecture is anticipated. Representative UIPs have to be implemented to prove the VUI concept and refine its foundations. Due to UIP rendering needs of the VUI, the non UIP-based UI-Control compositions can benefit from the platform-independent rendering, too. In the end, the *AT* software character of *View Definition* components may be completely avoided.

The primary limitation of a VUI based solution will be its dependence on a strong OO language. One can argue that a VUI architecture is hard to establish for web-clients relying on browser based languages, such as JavaScript and popular frameworks like JQuery, due to lacking object-orientation. Frameworks like GWT [30] that are able to accept OO code and compile it to JavaScript may be a promising option for a VUI but can be limited due to the available set of UI-Controls. In the end, the CUI based code would need further enhancements to represent alternative definitions of UIPs currently not covered by present UI-Controls.

Finally, a VUI based approach will not be achieved without obstacles. The abstraction of common GUI framework operations to CUI level code for reuse by UIP definitions is not an easy task. Moreover, the design of interfaces and their operations has to suit current and future UIP definitions. The software category tree will help us to limit framework dependencies and plan the distribution of responsibilities among components.

VII. Conclusion and Future Work

In the future, UIPs are likely to become complementary assets for reuse in comparison to design patterns [31]. With the incorporation of UIPs as valuable assets for the reuse of parts of the implementation code, the complexity of GUI artifacts to be designed and developed manually would be reduced. Much of the former GUI programming would be replaced by configuration of chosen UIP instances. As a consequence, the developers could focus more on application relevant design. However, current approaches that employ

 ISBN: 978-1-61208-367-4

UIPs on the basis of model-based generation are still challenged by formalization issues and have not proven their UIP variability concepts yet.

Future work. The alternative VUI based approach will be further elaborated in our future work. At first, the common GUI design issues have to be solved by a detailed GUI reference architecture. On the basis of the presented software category models and our VUI draft, we will be able to identify a suitable component based architecture. The requirements for a VUI based solution will be complemented by example UIPs and implementations. During that process, both category and UIP requirements models will be updated. Finally, we will investigate on the impacts of UIPs on other architecture artifacts and their traceability connections.

REFERENCES

[1] J. Tidwell, Designing Interfaces. Patterns for Effective Interaction Design. Beijing: O'Reilly, 2006.

[2] D. Ammon, S. Wendler, T. Kikova, and I. Philippow, "Specification of formalized software patterns for the development of user interfaces," The Seventh International Conference on Software Engineering Advances (ICSEA 12) IARIA, Nov. 2012, pp. 296-303, ISBN: 978-1-61208-230-1.

[3] S. Wendler, D. Ammon, I. Philippow, and D. Streitferdt "A factor model capturing requirements for generative user interface patterns," The Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 13), IARIA, May 27 - June 1 2013, pp. 34-43, ISSN: 2308-3557.

[4] F. Radeke and P. Forbrig, "Patterns in task-based modeling of user interfaces," The Sixth International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 07), Springer LNCS 4849, Nov. 2007, pp. 184-197.

[5] J. Engel and C. Märtin, "PaMGIS: A framework for pattern-based modeling and generation of interactive systems," The Thirteenth International Conference on Human-Computer Interaction (HCII 09), Part I, Springer LNCS 5610, July 2009, pp. 826-835.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stahl, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. New York: John Wiley & Sons, 1996.

[7] M. Fowler, Patterns of Enterprise Application Architecture. New Jersey: Addison-Wesley Professional, 2003.

[8] M. Fowler, "Development of Further Patterns of Enterprise Application Architecture," http://martinfowler.com/eaaDev/index.html, 2014.08.14.

[9] M. Haft, B. Humm, and J. Siedersleben, "The architect's dilemma – will reference architectures help?," First International Conference on the Quality of Software Architectures (QoSA 2005), Springer LNCS 3712, Sept. 2005, pp. 106-122.

[10] S. Wendler, "A software category model for graphical user interfaces," The Ninth International Conference on Software Engineering Advances (ICSEA 2014), IARIA, in press.

[11] S. Siedersleben, Ed., Softwaretechnik: Praxiswissen für Software-Ingenieure [Software engineering: practical knowledge for software engineers], 2nd ed. München: Hanser, 2003.

[12] S. Wendler, D. Ammon, T. Kikova, I. Philippow, and D. Streitferdt, "Theoretical and practical implications of user interface patterns applied for the development of graphical user interfaces," International Journal on Advances in Software, vol. 6, nr. 1 & 2, pp. 25-44, 2013, IARIA, ISSN: 1942-2628, http://www.iariajournals.org/software/.

[13] S. Wendler and D. Streitferdt, "An analysis of the generative user interface pattern structure," International Journal On Advances in Intelligent Systems, vol. 7, nr. 1 & 2, pp. 113-134, 2014, IARIA, ISSN: 1942-2679, http://www.iariajournals.org/intelligent_systems/index.html.

[14] J. Siedersleben and E. Denert, "Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur [How to build information systems? Thoughts on a standard architecture]," Informatik Spektrum, vol. 23, issue 4, Aug. 2000, pp. 247-257, doi: 10.1007/s002870000110.

[15] J. Siedersleben, Moderne Softwarearchitektur [Modern software architecture], 1st ed. 2004, corrected reprint. Heidelberg: dpunkt, 2006.

[16] S. Alpaev, "Applied MVC patterns. A pattern language," The Computing Research Repository (CoRR), May 2006, http://arxiv.org/abs/cs/0605020, 2014.08.14.

[17] A. Karagkasidis, "Developing GUI applications: architectural patterns revisited," The Thirteenth Annual European Conference on Pattern Languages of Programming (EuroPLoP 2008), CEUR-WS.org, July 2008.

[18] A. Dearden and J. Finlay, "Pattern languages in HCI; A critical review," Human-Computer Interaction, vol. 21, issue 1, 2006, pp. 49-102.

[19] S. Wendler and I. Philippow, "Requirements for a definition of generative user interface patterns," The Fifteenth International Conference on Human-Computer Interaction (HCII 13), Part I, Springer LNCS 8004, July 2013, pp. 510-520.

[20] A. Wolff, P. Forbrig, A. Dittmar, and D. Reichart, "Tool support for an evolutionary design process using patterns," Workshop on Multi-channel Adaptive Context-sensitive Systems (MAC 06), May 2006, pp. 71-80.

[21] UsiXML website, http://www.usixml.org/, 2014.08.14.

[22] K. Breiner, K. Bizik, T. Rauch, M. Seissler, G. Meixner, and P. Diebold, "Automatic adaptation of user workflows within model-based user interface generation during runtime on the example of the smartmote," The Fourteenth International Conference on Human-Computer Interaction. Design and Development Approaches. (HCII 2011), Part I, Springer LNCS 6761, July 2011, pp. 165-174.

[23] K. Breiner, G. Meixner, D. Rombach, M. Seissler, and D. Zühlke, "Efficient generation of ambient intelligent user interfaces," The Fifteenth International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 11), Springer LNCS 6884, Sept. 2011, pp. 136-145.

[24] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml, 2014.08.14.

[25] G. Meixner, M. Seissler, and K. Breiner, "Model-Driven useware engineering," Model-Driven Development of Advanced User Interfaces, Studies in Computational Intelligence, vol. 340, H. Hussmann, G. Meixner, and D. Zuehlke, Eds., Berlin, Heidelberg: Springer, pp. 1-26.

[26] K. Bittner and I. Spence, Use case modeling, 8th print. Boston (Mass.): Addison-Wesley, 2006.

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

[28] M. van Welie, "A pattern library for interaction design," http://www.welie.com, 2014.08.14.

[29] J. Vanderdonckt, "A MDA-compliant environment for developing user interfaces of information systems," The Seventeenth International Conference on Advanced Information Systems Engineering (CAiSE 2005), Springer LNCS 3520, June 2005, pp. 16-31.

[30] Google Web Toolkit, http://www.gwtproject.org/, 2014.08.14.

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

 ISBN: 978-1-61208-367-4