Stefan Wendler, Danny Ammon, Teodora Kikova, Ilka Philippow, Detlef Streitferdt

Theoretical and practical implications of user interface patterns applied for the development of graphical user interfaces

Original published in:

International Journal on Advances in Software. - [S.l.] : IARIA. – 6 (2013), 1&2, p. 25-44.

ISSN: 1942-2628

URL: http://www.iariajournals.org/software/tocv6n12.html

[Visited: 2015-01-28]

Theoretical and Practical Implications of User Interface Patterns Applied for the Development of Graphical User Interfaces

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

Abstract — **We address current research concerning patterns dedicated to enable higher reusability during the automated development of GUI systems. User interface patterns are promising artifacts for improvements in this regard. Both general models for abstractions of graphical user interfaces and user interface pattern based concepts such as potential notations and model-based processes are considered. On that basis, the present limitations and potentials surrounding user interface patterns are to be investigated. We elaborate what theoretical implications emerge from user interface patterns applied for reuse and automation within user interface transformation steps. For this purpose, formal descriptions of user interface patterns are necessary. We analyze the capabilities of the mature XML-based user interface description languages UIML and UsiXML to express user interface patterns. Additionally, we experimentally investigate and analyze strengths and weaknesses of two general transformation approaches to derive practical implications of user interface patterns. As a result, we develop suggestions on how to apply positive effects of user interface patterns for the development of pattern-based graphical user interfaces.**

Keywords — graphical user interface development; model-based software development; HCI patterns; user interface patterns; UIML; UsiXML

I. Introduction

Interactive systems. Interactive systems demand for a fast and efficient development of their graphical user interface (GUI), as well as its adaptation to changing requirements throughout the software life cycle. In this paper, E-Commerce software serves as a representative of these interactive systems. Currently, these are a fundamental asset of modern business models providing B2C interaction via online-shops. In many cases, such systems are offered as standard software, which allows several customization options after installation. In this context, they are differentiated into the application kernel and a GUI system.

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

This is a revisited and substantially augmented version of "Development of Graphical User Interfaces based on User Interface Patterns", which appeared in the in Proceedings of The Fourth International Conferences on Pervasive Patterns and Applications (PATTERNS 2012) [1].

Limited customizability of GUIs. Contrary to the application kernel, the customization of the GUI is possible only with rather high efforts. An important reason is that software patterns do not cover all aspects needed for GUIs. These patterns have been commonly applied for GUIs [2][3], but in most cases they are limited to functional and control related aspects [4]. The visual and interactive components of the GUI are not supported by software patterns yet. Furthermore, the reuse of GUI components, e.g., layout, navigation structures, choice of user interface controls (*UI-Controls*) and type of interaction, is only sparsely supported by current methods and tools. For each project with its varying context, those potentially reusable entities have to be implemented and customized anew, leading to high efforts.

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

User interface patterns. Along with other researchers [5] [6] [7] [8] [9], we propose an approach to this problem through the deployment of User Interface Patterns (UIPs). These patterns offer well-proven solutions for GUI designs [10], which embody a high quality of usability [11]. So far, UIPs usually have not been considered as source code artifacts, in contrast to software patterns. Current UIPs and their compilations mostly reside on an informal level of description [5]. The research towards formal pattern representations is still in progress.

A. Objectives

In this paper, we elaborate that formal UIPs can assist in raising effectiveness and efficiency of the development process of a GUI system. For a start, we present and analyze conceptual models for the GUI development to valuate and position UIPs as unique artifacts. In this regard, we describe, from a theoretical point of view, how reuse and automation within GUI transformation steps can be established by the deployment of UIPs.

Moreover, we present and review current approaches concerning the definition, formalization, and deployment of UIPs within model-based software development processes dedicated to GUI-systems. On this basis, we discuss the limitations and possibilities of transformations into executable GUIs. For that purpose, two different transformation approaches have been experimentally investigated. These approaches will be assessed facing two different GUI dialog examples. As a result, we derive practical implications of UIPs and develop suggestions, how the positive effects of UIPs for the development of GUIs can be applied. Finally, influences resulting from the use of UIPs in the development process are discussed.

B. *Structure of the Paper*

In Section II, selected state of the art and related work according to general applicable models for the GUI development are presented. The next section is dedicated to the current state of concepts and processes already applying UIPs as software artifacts. Both parts of related work are assessed according to our objectives in Sections IV and VI respectively. Subsequently, the theoretical implications of UIPs on the development process for GUIs are elaborated in Section V. Afterwards, Section VII presents our two approaches for the transformation of formal UIPs into source code. The practical implications of UIPs resulting from their application in experimental transformations are presented in Section VIII, which also combines the findings of Sections V and VII for discussion. Finally, our conclusions are drawn and future research options are outlined in Section IX.

II. Related Work: GUI Development Processes

The development of GUI systems still remains a challenge in our days. To discuss the activities and potentials of UIPs independently from specific software development processes and requirement models, we refer to generic model concepts. In the following sub-sections, we present two models, which describe activities and capture work products of the GUI specification process. Additionally, an early generation concept for GUI systems is presented.

A. *GUI Specification Process and Model Transformations*

A general GUI specification model. In reference [12], Ludolph elaborates the common steps of a GUI specification process. To master the complexity that occurs when deriving GUI specifications from requirement models, Ludolph proposes four model layers and corresponding transformations built on each other. Three of them, being relevant in our context, are depicted in Figure 1.

Essential model. By the *essential model*, all functional requirements and their structures are described. This information consists of the core specification, which is necessary for the development of the application kernel. Examples for respective artifacts are *use cases*, domain models and the specification of *tasks* or functional decompositions. These domain-specific requirements are abstracted from the realization technology, and thus, from the GUI system [12].

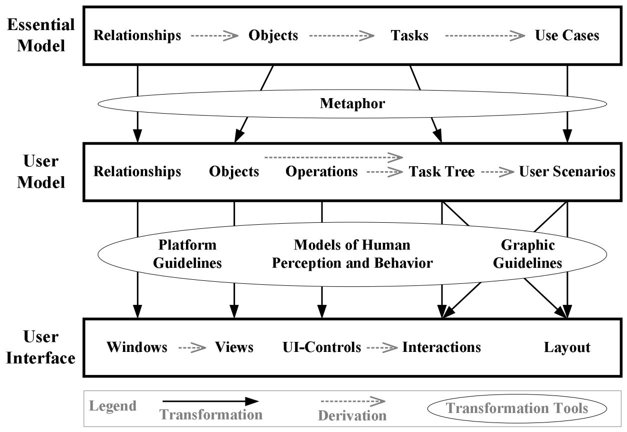

Figure 1. Model transformations of the GUI development process based on [12]

Consequently, a GUI specification must be established to bridge the information gap between requirements and a GUI system.

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

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

User interface. This model is used for establishing the actual GUI specification. Through the three parts rough layout, interaction design and detailed design [12], the appearance and behavior of the GUI system are concretized. The aim is to set up a suitable mapping between the elements of the *user model* and *views*, *windows*, as well as *UI-Controls* of the *user interface*. For the *metaphors* chosen before, graphical representations are now to be developed. The *objects* to be displayed, their attributes and the relations between them are represented by *views*. Subsequently, the *views* are arranged in *windows* according to the activities of the *user scenarios*, or alternatively, to the structure of the more detailed *task trees*. In these steps, there are often alternatives, which are influenced by style guides or the used GUI library and especially by the provided *UI-Controls*. At the same time, generic interaction patterns are applied as transformation tools, which also have an impact on the choice of *UI-Controls*.

B. Cameleon Reference Framework

User interface challenges. In reference [13], Vanderdonckt presents a GUI specification and development model, which is more concerned with handling environmental and non-functional requirements of GUI systems. The challenges to overcome are represented by different user skills and cultures. In addition, a user interface should be aware of different usage contexts and respective user intentions as well as working environments and individual capabilities of devices the user interface is running on.

Need for automation. GUI development is tedious when facing the above mentioned challenges, and thus, Vanderdonckt states in [13] that normally, GUIs would have to be developed for each context or device separately. A reason is given by the difficulty to source common or shared parts of the user interfaces. Since architectures and final code or frameworks have a great impact on the final shape of the certain user interfaces, the potential reuse is largely limited. Finally, advice is given to employ model-driven software development techniques within a GUI development environment.

To approach a solution, which copes with both the challenges and need for model-driven development, Vanderdonckt proposes a methodology, which consists of GUI modeling abstractions or steps besides a method and tool support. The proposed four modeling steps [13], originated from [14], are described in the following paragraphs:

Task & Concepts (T&C). The *tasks* to be performed by the user, while interacting with the GUI-system, are specified during this step. Additionally, *domain concepts* relevant to those *tasks* are specified as well.

Abstract UI (AUI). With the AUI, *tasks* are being grouped and structured by Abstract Interaction Objects (AIOs): *Individual Components* and *Abstract Containers* are both sub-types of AIOs and form the main elements of an AUI. These resemble rather abstract entities serving for definition and structuring purposes only. Thus, AIOs come without any technical appearance or other format of imagination, since the options to shape them are very different during the next two modeling steps and should be preserved for developers. Besides the structuring of AIOs, an AUI specifies very basic interaction information such as input, output, navigation and control [5], which is defined independently from modality. Finally, the AUI acts as a "canonical expression of the rendering of the domain concepts and tasks" [13].

Concrete UI (CUI). The CUI refines the elements of an AUI to a complete but platform-independent user interface model. In this regard, Concrete Interaction Objects (CIOs) refine the AIOs of the AUI. CIOs resemble a chosen set of both *UI-Controls* or containers and their respective properties. While resembling an abstraction, the CUI "abstracts a FUI into a UI definition that is independent of any computing platform" [13].

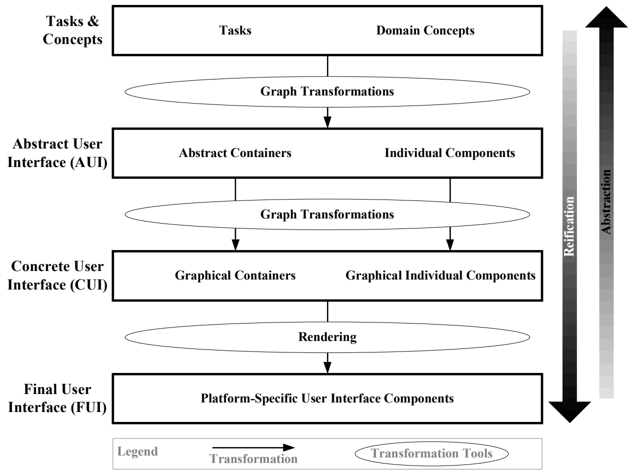

Figure 2. Modeling steps of the Cameleon Reference Framework based on [13] and implemented by UsiXML [15]

Final UI (FUI). As the last refinement, the FUI represents a certain device or platform specific user interface model. So, it embodies the final user interface components running in that specific environment.

The above described modeling steps are depicted by Figure 2, which is focused on graphical user interface implementations, as this is the case for its source [13].

UsiXML. To express the occurring models within these modeling steps, the GUI specification language UsiXML (user interface extensible markup language) [15] has been developed. Concerning the modeling facilities for the CUI step, UsiXML offers a specific set of CIOs sourced from common UI toolkits or frameworks. Therefore, the available modeling elements represent an intersection set of common GUI element sets.

C. Generators for graphical User Interfaces

To raise efficiency in GUI development, concepts and frameworks have been invented, which are able to generate complete GUI applications based on a partly specification of the application kernel or comparative model bases. Here, Naked Objects [16] and JANUS [17] can be mentioned. Both rely on an object-oriented domain model, which has to be a part of the application kernel. Based on the information provided by this model, standard dialogs are being generated with appropriate *UI-Controls* for the repetitive *tasks* to be carried out in conjunction with certain *objects.* For instance, to generate an *object* editor for entities like product or customer, certain text fields, lists or date pickers are selected as *UI-Controls*, which match the domain data types of the selected domain *object* for editing.

III. RELATED WORK: USER INTERFACE PATTERNS

In this part of related work, we present definitions, notations and concepts that address or employ patterns specific for model-based user interface development.

A. *User Interface Pattern Definition and Types*

Current research has been discussing Human-Computer-Interaction (HCI) patterns [18] and especially User Interface Patterns (UIPs) for a longer period [19] [5] now. A UIP can be defined as a universal and reusable solution for common interaction and visual structures of GUIs. UIPs are distinguished by two types according to Vanderdonckt and Simarro [5]:

Descriptive UIPs. Primarily, UIPs are provided by means of verbal and graphical descriptions. In this context, UIPs are commonly specified following a scheme similar to the one used for design patterns [20]. By reference [21], a specialized language for patterns was proposed, which is named PLML (pattern language markup language). Details about the language structure can be found in [22] as well as its XML DTD in [5]. A practical application of its descriptive capabilities for several types of patterns, which may occur in conjunction with the Cameleon Reference Framework, is also outlined in [5].

UIP-Libraries. UIP libraries such as [23], [24], and [25] provide numerous examples for descriptive UIPs. Based on the presented categories, concepts about possible UIP hierarchies and their collaborations can be imagined.

Formal UIPs. Generative UIPs [5] are presented rarely. In contrast to descriptive UIPs, they feature a machine-readable form and are regarded as formal UIPs accordingly. The format for storing such UIPs may constitute of a graphic, e.g., UML [19] or XML based notation [26] [8] [9]. The formal UIPs are of great importance, since they can be used within development environments, especially for automated transformations to certain GUI-implementations.

B. *Formalization of User Interface Patterns*

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

UsiXML and UIML. In our preparation, we conducted an extensive investigation on formal GUI specification languages and their applicability for UIPs. As result, two languages with an outstanding maturity have been identified.

Intentionally, the XML-based languages UsiXML [15] and UIML [27] were developed for specifying a GUI independently from technology and platform specifics. However, such languages may be applicable for UIPs. One the one hand, UIML offers templates and associated parameters for reusing pre-defined structures and behavior of GUI components. On the other hand, UsiXML is designed to implement the Cameleon Reference Framework, which already propagated higher reuse by its abstractions of GUI modeling steps as well as automated processing by model-driven software development techniques. Moreover, both indeed have been applied in model-based processes or have been extended for that context. More information on that is provided in Section III.C.

IDEALXML. To raise the efficiency of GUI development environments, tools are necessary that facilitate formal specifications of UIPs with regard to language definitions and rules. A widespread tool concept for UsiXML is presented with IDEALXML [13] [5]. By using the various models defined by UsiXML as an information basis, many aspects of a GUI and additionally the applied domain model of the application kernel are included in the GUI specification. As a result, a detailed and comprehensive XML specification for the GUI can be created. Many aspects of the *user model* from [12] are already included.

C. *Model-based Processes with User Interface Pattern Integration*

The pattern conception emerged from the HCI research has already been taken into consideration for model-based software development of GUI-systems. Researchers have introduced several model frameworks and notations to express generative UIPs, and thus, enable formalization facilities for descriptive UIPs. A common basis assumed for all different processes is a task based *user model* that is exploited to derive dialog and navigation structures of the user interface. Yet, all approaches have not reached a sufficient maturity level according to the available publications. They still were drafting or enhancing their processes, tools or notations as they had been by challenged relevant issues surrounding generative pattern definition and application.

Queen's University Kingston. Zhao et al. [6] proposed the detailed modeling of tasks in order to be able to group them into segments, which are being transformed to dialogs displaying the associated data or contained sub-tasks.

As challenges for future work, two main aspects remained: the evaluation of achieved usability by the pattern application and the extension of customization abilities of the underlying framework to allow the definition of specific *UI-Controls* and even patterns to be integrated into the established process were suggested in [28]. In addition, the integration of more user interface patterns along with guidelines for final UI design as well as an enhancement of the task analysis to exploit more information relevant for UI generation were outlined in [6] as future work.

University of Rostock. Radeke et al. [29] presented a modeling framework that would be capable of employing patterns for all involved models (task, dialog, presentation and layout). Since the approach was focused on task modeling and respective patterns, the derivation of dialog structures was a main outcome. In order to enhance their capabilities towards pattern application for CUI models, UsiPXML (user interface modeling pattern language) was introduced in [26] as a notation to express all kinds of involved patterns. Being based on UsiXML as well as PLML, the new notation incorporated enhancements like structure attributes and variables to allow for a context-specific instantiation of a defined pattern.

However, future challenges were stated as follows. The need for enhanced tool support and the definition of more complex patterns was raised in [30]. Moreover, the pattern representation on the CUI level with UsiXML should be revised as well according to [31]. Lastly, the expansion of the set of available patterns and the concept of pattern inter-relationships were relevant considerations in [26]. For the latter, the research question about how task and dialog patterns would influence other patterns situated on lower levels is left open.

University of Augsburg. An alternative modeling framework integrating patterns on selected model stages was suggested by Engel and Märtin in [8]. Rooted in principles on the structuring of pattern languages [32], the main emphasis was laid on the hierarchy of patterns and their notation [33], which was based on a custom XML DTD for the generative part.

For the encountered challenges, future activities were considered, which would enrich the implementation aspects of pattern descriptions [34] and deliver concepts of pattern relationships. In the focus of transformations, future work was seen for the derivation of concrete UI models from abstract ones [35].

University of Kaiserslautern. Starting with criticism of recent approaches of other researchers, Seissler et al. [9] proposed a third modeling framework with comparative models and patterns, but they employed different notations and introduced a suggestion for a classification of pattern relationships. Additionally, the need for runtime adaptation of user interfaces was considered [36] as well as the concept of encapsulation of UIML fragments [9] within their notation to express user interface patterns.

They emphasized on tool support for pattern instantiation or the adaptation of patterns to different contexts of use that may even change at runtime [36]. Moreover, a proper tool for pattern selection and integration as well as the refinement of inter-model pattern relationships were stated as future challenges in [9]. The latter was considered to reflect the relations between pattern of different abstractions in order to offer better modularization and provide options for patterns that may be better suited for a specific context. Finally, Seissler et al. recognized in [9] that their future work should extend the pattern language for further testing of their notation approach.

IV. Model Considerations for Deploying User Interface Patterns

This section is intended to discuss the first part of related work presented in Section II. Before the more advanced concepts of Section III are addressed, the transition of traditional GUI specification and development towards a pattern-based solution shall be attended to. In this context, we outline the possible deployment of UIPs in development processes referring to both conceptual models elaborated by Ludolph and Vanderdonckt.

A. *Review of the GUI Specification Model by Ludolph*

Model transformations as described by Ludolph [12] illustrate a detailed account of relevant model elements for the GUI specification of the covered domain. However, any transformations are carried out manually. Besides that, no automation and only few options for reuse are mentioned.

However, artifact dependencies are detailed and the transformation of *essential model* requirement elements to certain *user interface* model elements is outlined. For the final transformation, Ludolph suggests manual and cognitive means of transformation, which lead to clearly defined dependencies between *user model* and *user interface* entities. These prerequisites are ideal to be considered in the discussion on how UIPs influence artifacts. Particularly, it is of interest, how a GUI specification can be developed starting from a basis of functional requirement artifacts and using UIPs as bridging elements for transformations.

B. *Review of the Cameleon Reference Framework*

Relevance. From our point of view, the Cameleon Reference Framework as presented in [13] resembles a valuable model foundation or mental concept for UIPs, since it addresses the following two aspects. Firstly, GUI development activities and related tool support to decide on automation steps are covered. Secondly, pattern deployment possibilities and related abstractions may be derived. In this regard, a developer can decide on the granularity, reach and modularization of potential patterns while having the four segregated modeling steps on his mind. However, the latter aspect was not met by the original source and is only inspired.

GUI development aspect. As far as the first aspect is addressed, the proposed model abstractions or steps resemble UI concerns applicable to a wide range of different domains. The model abstractions make sense as they address the elaborated challenges in [13] by a separation of concerns. The four steps have been introduced to handle the various challenges or requirements by sharing or distributing them across the abstractions. Consequently, the separation of models enables different grades of reuse and an isolation of particular challenges, as they are no longer bound to single GUI models but to a set of models as proposed.

To approach the modeling steps, a strict top-down decomposition procedure is not required. In contrast, the entry point is variable so that one can start with an AUI or CUI without tasks modeling at all. A user interface may be subsequently abstracted or refined across the proposed reification model stages.

Moreover, the steps aid both in forward and reverse engineering, since they demand for explicitly capturing implicit knowledge applied in both model transformation paths: the refinement towards a FUI can be approached by subsequent increase in detail, which is stored in segregated models and their elemental notations. As the reification of an AUI towards CUI is progressing, the elementary concepts embodied by AIOs of different dialogs can be lined up to identify reoccurring structures. In this respect, AIOs are an abstraction and so they do share the commonalities of certain GUI structures. Consequently, identified AIO structures offer potentials to discover UIPs for the particular domain during the transition to the CUI.

Concerning reverse engineering, the abstraction of a given FUI or CUI model to abstract grouped tasks embodied by AIOs is also supported. The derived AUI may be reified to another platforms' CUI. If an AUI was already created by forward engineering, a modeling step could be avoided for the migration to other platforms or devices.

For practical implementation, transformation means or tools mentioned as in the Ludolph model are missing. Although the models used for implementing the four steps are closely related to the UsiXML language, the associated metamodel as a potential implementation is still work in progress. At usixml.org, no current version could be consulted. Therefore, no detailed mappings like in the Ludolph model could be depicted.

Pattern incorporation aspect. As respective implementations of the Cameleon modeling steps, the presented models in [13] and [5] currently do not outline the reuse or modularization of artifacts. A proper pattern-based view to overcome the manual "translation" [13] process between available models still has to be invented. At last, models or fragments of them can only be reused in their completeness and are not abstracted further. Patterns may be instantiated at various modeling steps (e.g., AUI, CUI) as suggested in [5], but can hardly be adapted to other contexts without manual re-modeling. To conclude, an additional abstraction inside modeling steps, which allows for pattern definition and instantiation, is missing and is not provided by the available sources.

As far as UIPs are concerned, these patterns should not be associated to the AUI, since the latter is too abstract for UIPs. Certain UI-Controls cannot be modeled or imagined on the AUI level, so that a great portion of an individual UIP's characteristics cannot be expressed. The resulting refinement work to "reify" [13] an AUI based UIP towards a CUI representation would denote a considerable effort. For instance, whenever a selection AIO is encountered inside a UIP definition, there would be more than one possible reification available like a combobox, listbox or a radio group. Therefore, it could be implied that the model-to-model transformation between AUI and CUI relied on extensive manual configuration or intervention, as the CUI does possess much more detail than the AUI. Otherwise, strict rules to enable automated graph transformation may prevent the expression of particular UIPs. Lastly, for the particular domain addressed here, UIPs rely on the WIMP (windows, icons, menus and pointer) paradigm, so AUI considerations will not merit extensive reuse as this would be permitted by a CUI model.

With respect to the CUI modeling stage, the applied notation like UsiXML would have to reflect a chosen set of *UI-Controls*, events and containers as well as their chosen set of properties. These sets may already limit the expressiveness of UIPs or an issue would be the integration of new types or properties. Due to the fact that particular UIPs may exclusively address certain devices or platforms or that other classifications of UIPs may restrict their reusability to a certain domain [37], even the CUI level would be too abstract to allow for an exact representation. If this aspect would not pose an issue in a certain development environment, UIPs clearly are to be settled on the CUI level, since there are several advantages for keeping UIPs on that particular abstraction level:

As mentioned in [13], a notation like UsiXML or even UIML could be used to express UIPs on the CUI level leading to the benefits of these languages. Firstly, for the machine-readable XML languages no programming skills would be needed. Secondly, with XML as a basis, the notation would posses a standard format and vast tool support (parsers, editors). Thirdly, "cross-toolkit development" [13] would be possible and UIP sources could be kept independently from changing GUI platforms or frameworks and lastly, programming languages.

C. *Exertion of Ludolph and Cameleon Models*

Current state of the art has proposed own specific model frameworks as mentioned in Section III.C. These approaches neither have achieved a truly reusable pattern-based solution yet, nor have they positioned UIPs in relation to generally applicable fundamentals. Since the transformations by Ludolph or the Cameleon model have been formulated from different perspectives, but still embody general concepts, we take them into consideration to derive theoretical and practical implications of UIPs.

Different focus. The model by Ludolph is focused on particular artifacts, their transformations and related measures. In contrast, the Cameleon Reference Model by Vanderdonckt presents abstractions to treat environments, devices, portability, and most notably, the software production environment, as XML and automation or model-driven software development are of the essence.

GUI transformations by Ludolph. The model established by Ludolph can be considered as a refinement of the *Tasks & Concepts* as well as the CUI level for graphic user interfaces, since most artifacts can be allocated to one of these levels. An AUI level is actually missing and only implicitly established by the augmentation of *user model* elements with *metaphors*. The final stage of the Ludolph model can be defined in terms of the CUI when specification notations like UIML or UsiXML-CUI are being used.

Cameleon. The Cameleon Reference Model is the more abstract model as its details are to be defined by the implementation language, especially by UsiXML, and the particular context of use or domain. Due to the defined modeling stages, pattern deployment and modularization concerns can be approached more gentle rather than being trapped in discussions of how to structure a pattern language for certain artifacts [38].

Shared limitation. Both models do not feature a clearly distinguished pattern dimension.

Reuse may be already addressed by Ludolph for GUI structures within a certain project. For instance, the *views* associated to certain *objects* may experience reuse in each *task* they are handled by different operations. However, *objects* tend to change in the face of different contexts, domains, users and thus, real pattern-based reuse across different projects is missing.

Although the pattern support for the UsiXML metamodel was already inspired by Vanderdonckt as a "Translation" [13] of models to different contexts and PLML-patterns in the environment of IDEALXML [5], it has not been implemented in the main language facilities of UsiXML yet.

Exertion. The model by Ludolph is already detailed concerning domain artifacts. Therefore, it will be used to discuss both the theoretical and practical implications of UIPs on artifact development stages. Nevertheless, it is not suitable to position UIPs without the conception of a pattern language or hierarchy. Märtin et al. [32] [33] support a fine-grained structure, which is clearly neglected by Seissler et al. [9]. Furthermore, pattern relations are still to be outlined in most model-based approaches as mentioned in Section III.C. Assuming that a pattern language with appropriate pattern relationships would have been elaborated, Ludolph's model may be customized for the particular domain, as it already holds artifacts typical for business information systems.

The Cameleon Reference Framework will be taken into consideration to position UIPs concerning their practical implications. In this context, the abstraction level of UIPs has to be discussed, i.e., how concrete UIPs should be compared to implementation level GUI elements. Additionally, technical considerations should be addressed like the coupling to GUI frameworks and programming languages. The most important fact is the positioning of UIPs in the light of potential notations, which have been introduced in Section III.B.

D. *Limitations of GUI-Generators*

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

Limited functionality. The information for generating the GUI is restricted to a domain model and previously determined dialog templates along with their *UI-Controls*. Hence, their applicability is limited to operations and relations of single domain *objects*. When multiple and differing domain *objects* do play a role in complex *user scenarios* [12], the generators can no longer provide suitable dialogs for the GUI application. Moreover, extensive interaction flows require hierarchical decisions, which have to be realized, e.g., by using wizard dialogs. In this situation, GUI generators cannot be applied. The connection between dialogs and superordinate interaction design still has to be implemented manually.

Uniform visuals. A further weakness is related to the visual GUI design. Each dialog created by generators is based on the same template for the GUI-design. Solely the contents which are derived from the application kernel are variable. Both *layout* and possible *interactions* are fixed in order to permit the automatic generation. The uniformity and its corresponding usability have been criticized for Naked Objects [39]. Assuming the best case, the information for GUI design is based on established UIPs and possesses their accepted usability for certain *tasks*. Nevertheless, the generated dialogs look very similar and there is no option to select or change the UIPs incorporated in the GUI design.

V. Theoretical Implications of User Interface Patterns

In this section, the theoretical implications of UIPs are derived on the basis of considered models of Section IV and the following scenario serving as a background.

A. *Application Scenario: GUI Customization of Standard Software*

On the basis of the customization of GUIs for standard software and the model transformations described in Section II.A, the theoretical implications of UIPs are to be considered. To present an example of standard software, we refer to e-commerce software, which usually offers both a front-end system for online-shopping and a back-end system to manage orders and stock.

Common essential model. This kind of standard software fulfils the functional requirements of a multitude of users at the same time. Therefore, these systems share a well-defined *essential model* that specifies their functional range and has many commonalities along existing installations. Standard software implements the *essential model* through different components of the *Application Kernel* as shown in Figure 3. Each installation consists of a configuration for the *Application Kernel*, which includes many already available and little custom components in most cases. In this context, the *User Interface* acts as a compositional layer that combines *Core* and *Custom Services* together with suitable dialogs for the user.

Individual GUIs for eShops. Concerning eShops, the visual design of the GUI is of special relevance, since the user interface is defined as a major product feature that differentiates the competitors on the market. Hence, the needs of customers and users are vitally important in order to provide them with the suitable and individual dialogs. In this regard, the proportions of components related to the whole system are symbolized by their size in Figure 3.

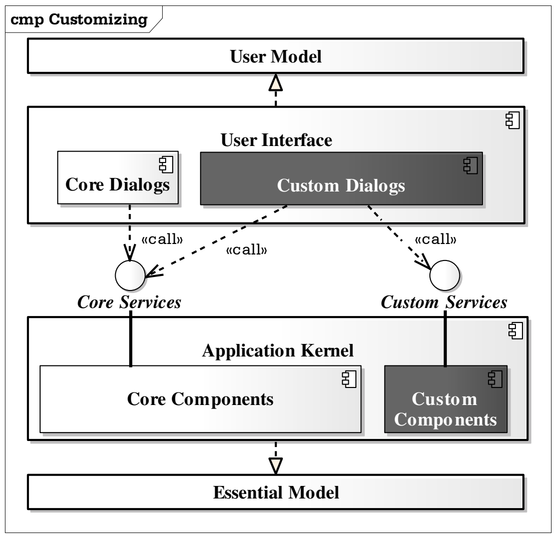

Figure 3. Components involved in the customization of standard software

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

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

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

B. Model Aspects of User Interface Patterns

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

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

Since UIPs are defined more extensively and concretely, they can be applied as a transformation tool instead of using *metaphors*. Descriptive UIPs feature a pattern-like description scheme that, for example, is provided in the catalogues in [23] and [24]. Thus, they offer much more information and sometimes even assessments, which can inspire the GUI specification. In addition, descriptive UIPs do already possess visual designs that may be exemplary, or in the worst-case, abstract.

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

When UIPs are used in place of *metaphors* for formalization, these new entities can be integrated in the tools for specifications. Concerning UsiXML, UIPs could describe the CUIM. *Task-Trees* are already present in UsiXML, so this concept of specification partly follows the modeling concepts in [12] and thus may be generically applicable.

User model and UIPs. With regard to the *user model*, the numerous modeling steps no longer need to be performed with the introduction of UIPs. Instead, it is sufficient to derive the *tasks* from the *use cases* within the *essential model* and allocate UIPs for these. Detailed *task-trees* no longer have to be created, since UIPs already contain these operations within their interaction design. Nevertheless, *tasks* have to provide a certain level of detail to derive navigation structures [29].

Interactions can already be specified in formal UIPs. Later on, this information can directly be used for parts of the presentation control of *views* or *windows*. As a result, an extensive *user scenario* also is obsolete, as it was originally needed for deriving the more detailed *task-tree*. Now it is sufficient to lay emphasis on expressing the features of UIPs and their connection to the *tasks* defined by the *essential model*. The *objects* are also represented within the UIPs in an abstract way. With the aid of placeholders for certain domain data types, adaptable *views* for *object* data can already be prepared in formal UIPs. Finally, much of the afore-mentioned information of the *user model* now will be provided by completely specified UIPs.

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

For *interactions*, the originally applied *Models of Human Perception and Behavior* of Figure 1 are no longer explicitly needed since they are implicitly incorporated in the interaction designs of the UIPs. In this context, suitable types of *UI-Controls* are already determined by UIPs. Nevertheless, a complete and concrete GUI-design will not be provided by UIPs, since the number, ordering and contents of *UI-Controls* depend on the context and have to be specified by the developer with instance parameters accordingly. In the same way, *Platform* and *Graphic Guidelines* act as essential policies to adapt the UIPs to the available GUI-framework and its available *UI-Controls*.

Conclusion. We explained that UIPs might cover most parts of the *user model* as well as numerous aspects of the *user interface*. By using UIPs in the modeling process, these specification contents can be compiled based on the respective context without actually performing the two transformations from Figure 1 explicitly. Basically, the transformation to the target platform remains as depicted in Figure 4.

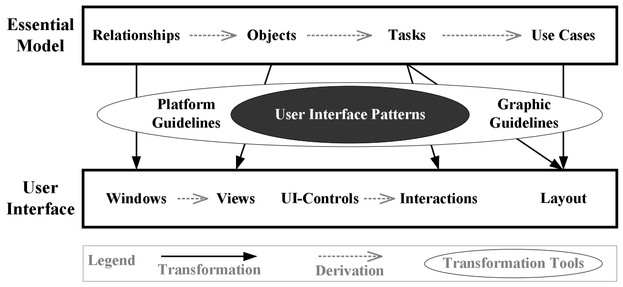

Figure 4. GUI transformations with the aid of UIPs and automation

C. *Influence of User Interface Patterns on GUI-Transformations*

In this section, the potentials of UIPs related to the GUI specification process are summarized from a theoretical perspective.

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

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

Many interaction designs can be derived from initial thoughts about GUI design for the most important *use cases* and their corresponding *tasks*. When a first UIP configuration has been created, the realization of the *Graphic* and *Platform Guidelines* therein can be adopted for other UIP-applications since the target platform is the same for each dialog of a system. Especially when *user scenarios* overlap, meaning they partly use the same *views* or *windows* as well as *object* data, UIPs enable a high grade of reuse. UIP assignments, already established for other *tasks,* can be reused with the appropriate changes.

E-commerce software tends to use many application components together although they are offered by different dialogs as illustrated in Figure 3. UIPs can contribute to a higher level of reuse in this context. Depending on the possible mapping between *Application Kernel* components and UIP-hierarchy, new dialogs can be formed by combining the views of certain services which are determined by their assigned UIPs.

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

Technically independent essential model. It is a common goal to keep elements of the *essential model* free from technical issues. Thus, the *essential model* has no reference to the GUI specification. Therefore, it is not subject to changes related to new requirements, which the user may incorporate for the GUI during the lifecycle of the system. User preferences often tend to change in terms of the visuals and *interactions* of the GUI. Concerning *use cases*, this rule of thumb is elaborated in [40] and [41]. Technical aspects and in particular the GUI specification are addressed in separate models such as *user model* and *user interface* according to [12]. After changes, these models have to be kept consistent what results in high efforts. For instance, a new or modified step within a *use case* scenario has to be considered in the corresponding *user scenario*, too.

By assigning UIPs to elements of the *essential model*, explicit *user models* and the prior checking of consistency between these models both become obsolete. Instead, *user models* will be created dynamically as well as implicitly by an actual configuration of UIPs and *essential model* mapping. The approach of Zhao et al. [6] strictly follows this concept. A technical transformation to the source code of the GUI that relies on the concrete appearances of the UIPs remains as shown in Figure 4. By modeling assignments between UIP and *task* or between UIP and *object*, the number of *UI-Controls*, the hierarchy and *layout* of UIPs, sufficient and structured information on the GUI system is provided. Subsequently, a generator will be able to compile the GUI suited for the chosen target platform. These theoretical influences enable an increased independence from the technical infrastructure, since the generator can be supplied with an appropriate configuration to instantiate the UIPs compatible to the target platform and its specifics.

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

VI. Review of UIP Notations and Applications

In this section, both potential notations and applications of UIPs are reviewed.

A. *Review Criteria for XML GUI Specification Languages*

Both languages are to be assessed by the following criteria:

Pattern variability criterion. The main criterion to be supported by a formalization language is the ability to allow the developer to abstract certain model structures to patterns. Each pattern embodies some points of variability to express a solution that is applicable and adaptable to a number of contexts. For instance, Figure 5 displays on the upper right hand side two exemplary UIP sketches. On the lower left hand side of Figure 5 possible UIP applications are drafted.

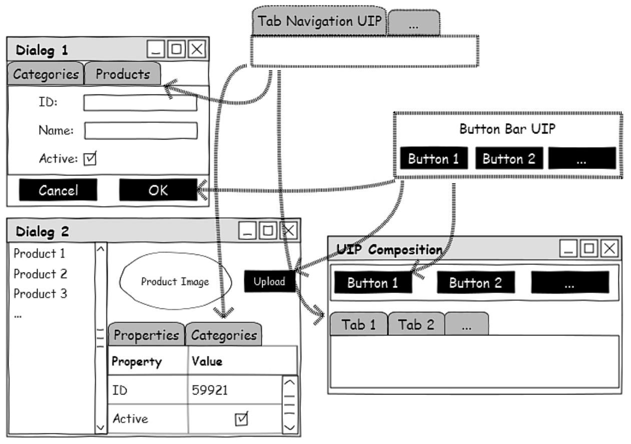

Figure 5. Schematic UIP examples and instances used in GUI dialogs

An apparent variability point of each illustrated UIP is the number of elements of the defined structure, e.g., how many buttons will appear in a certain UIP instance.

Content criteria. Besides the pattern abstraction criterion, three additional criteria are relevant for UIPs to be formalized. Firstly, the visuals to appear in the pattern structure have to be specified. In some cases certain *UI-Control* types make up the main impact of a certain UIP. For instance, the patterns "Collapsible Panels", "Carrousel", "Fly-out Menu" and the "Retractable Menu" sourced from [23] require certain *UI-Controls* that enable animation effects. It is important for the formalization to express *UI-Controls* that enable the desired *interaction* as close as possible while retaining a CUI level specification. Secondly, the layout of modeled structures has to be defined. Thirdly, stereotype behavior that is represented by the UIP has to be expressed.

B. *UsiXML User Interface Pattern Abstraction Capability*

Issues. The assessment of UsiXML is not an easy task compared to UIML. This is due to the facts that UsiXML is a far more complex language supporting most levels of Cameleon and it is not documented by a comprehensive specification with integrated examples as this is the case for UIML. At the time of writing, only older metamodels [42] of UsiXML and the W3C submission [43] of the AUI model [44] were available, possibly not reflecting new features.

Variability points. At its current state, whenever a pattern is to be expressed in UsiXML CUI, the variability points have to be avoided and specified directly. More precisely, it is only possible to specify a certain button bar or tab navigation instance with UsiXML. As far as we know, there is no way to parameterize the number of desired buttons or tabs. Thus, the described user interface structure looses on genericity [5]. Only the generativity [5] for a certain context and the platform- or device independence of the pattern remains on the CUI model of UsiXML. Other variability points for behavior and layout may be identified and reviewed. Unfortunately, this basic variability concern is a knock-out criterion.

IDEALXML. According to IDEALXML and its pattern expression capabilities [5], it was not mentioned how UIPs are being expressed in models such as the AUI or CUI model as reusable artifacts. Thus, it seems the patterns being modeled with the IDEALXML environment are always special instances to be manually adapted to new or changing contexts.

AUI patterns. Nevertheless, the AUI model and IDEALXML tool still might be mighty assets for pattern formalization. Following this thought, a developer would have to create AIOs of desired facets to model certain portions of a pattern, e.g., a single control facet for the button bar UIP or a single navigation facet for the tab navigation UIP of Figure 5. The modeling would solely be based on abstract structuring and interaction definition, as there would be no visual impressions of the final user interface. Later on, the instantiation of an AUI model pattern towards a CUI model would be prone to demand for fine-grained information, as each AIO would have to be configured individually to represent a specific set of CIOs and thus *UI-Controls*. In addition, language facilities would be needed to determine if an AIO was to be instantiated once or several times for a CUI. In any case, the modeling of UIPs with the UsiXML AUI model does not seem to be practical feasible, since user interface engineers would have a hard time to imagine the results. Finally, UIPs from public or corporate libraries could not be modeled with an adequate level of detail with respect to content criteria introduced in the previous section.

C. *UIML User Interface Pattern Abstraction Capability*

Reuse by templates. The UIML language facilities may enable the storing of UIPs. More precisely, UIML provides templates for the integration and reuse of already defined structures in new GUI formalizations [45]. The templates even may be parameterized, hierarchically nested and incorporated in the same way as ordinary <part> or <structure> elements [45]. Additionally, UIML templates may be used to restructure present <part> elements within a UIML document by the mechanisms of replace, union and cascade [45].

Sourcing of templates. UIML templates can only be sourced by concrete UIML structures, e.g., an existing <structure> or <part> element. The final element that incorporates any template must define certain values per <template-parameters> tag, which holds constants for the parameters of sourced templates [45].

Variability points. For UIPs to be stored inside a UIML document variability points need to be maintained. Therefore, it would be necessary to nest templates up to the structure root. In other words, the resulting main UIML document would have to resemble another template itself.

In this regard, even parameterized templates do not seem to be able to store UIPs deployable for varying contexts, since the respective parameters would have to be provided in the main UIML document. Unfortunately, a main UIML specification cannot be defined as a template that incorporates other templates and defines their variability point parameters, which would govern the elements of child templates. In detail, it is not allowed for <structure> to define parameters on that root level. Neither <interface>, <structure> or <part> tags can define own parameters to be processed by a pattern instantiation wizard [29] or similar tool.

Separation of instances and templates. To resolve this issue, a separation of UIML document types could be attempted where UIP definition and UIP instantiation are segregated. The UIML templates stand alone as separate files and may promise some reuse. Those templates can be sourced from the same or other UIML files. However, there are some restrictions as follows. As stated in the UIML 4.0 specification [45], <part> tags can only source <part>-based templates and <structure> tags <structure>-based templates respectively. Possible scenarios, which can be derived from this approach, are explained in the following sub-sections.

1) *Sourcing several <part>-based Templates*

In this approach, several UIML documents would each specify a certain UIP with (hierarchical) templates and respective parameters, repeated parts and maybe restructuring actions or behavior as additional options. A schematic example for this kind of solution related to the tab UIP and "Dialog 1" of Figure 5 is provided by Figure 6.

Definition of placeholders. As shown on the right hand side of Figure 6, one major UIML document would have to define the particular UIP instance or complete dialog ("Dialog 1") to be rendered. Separate container elements would have to be defined in the main UIML document serving as placeholders to be merged with the sourced template by either the replace, union or cascade options. In this regard, template parameters of UIML reside on the child node level as outlined on the right hand side of Figure 6. This implies that concrete parameters have to be passed to included templates and consequently, the final UIML document describing the UIP instances would have to be created for each application or dialog separately. In this way, the UIP instance document would be sourcing several smaller templates as lower level hierarchy <part> elements within their <structure>.

Separate definition of individual UIP instances. Finally, parameters would have to be provided and kept in the UIML UIP instance document as shown in Figure 6. Therefore, each UIP instance would have to be specified at root node level separately. The main UIML document would have to define the panels or containers to include UIPs into the hierarchy of the virtual tree. This is due to the fact that UIML template parameters may only be applied for root and child node level.

2) *Sourcing nested <part>-based Templates*

The reuse of several <part>-based templates could be approached, but contained structures would build a strict hierarchy. As depicted on the left hand side of Figure 6, for <part>-based templates only one root level container would be possible, which combines several nested <part> elements into the same sub-tree. Hence, the incorporation of two UIPs at the same time would result in a "virtual tree" [45] with equally ranked or nested elements inside the same container. The main UIML document could only source both UIPs within this strictly defined hierarchy and thus, the developer would replace a <part> with both UIPs at once. According to Figure 5, the tab UIP would be directly followed by the button UIP inside the same panel and the dialog data contents would be situated at the bottom differing from the actual desired layout depicted in Figure 5.

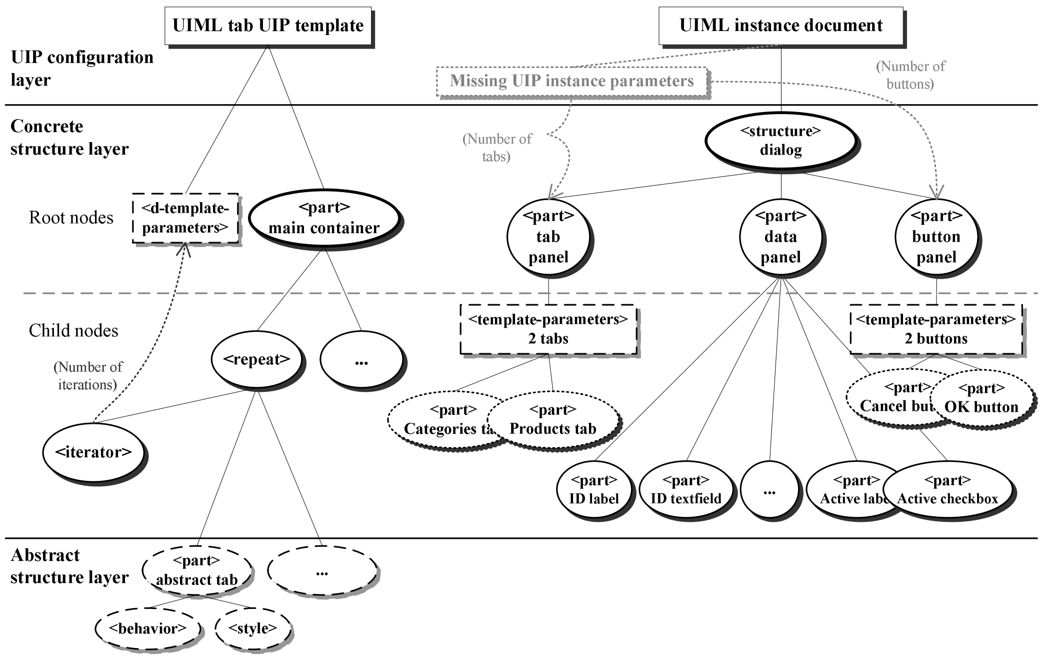

Figure 6. Schematic UIML <part>-based template and its sourcing inside a UIML document

3) *Sourcing <structure>-based Templates*

UIP compositions. Complex UIPs or their compositions like in Figure 5, forming entire new UIP units of reuse, could be specified with <structure>-based templates and hierarchical <part> elements. Following this approach, parameters could be applied to denote the iterators for each <part> at root node level included in the <structure>-based template. This variant is illustrated in Figure 7. Additionally, the cascade merging strategy could be used to preserve elements not to be replaced and the main UIML document would have to maintain a similar naming for <part> elements to be replaced by the template. In Figure 7, the <part> elements of both the template and UIML instance document are named equally.

However, these kinds of templates can only replace, cascade or union with one main <structure> element. Finally, this implies that only one template can be included in a UIML document using union or cascade at once. There is no sourcing of multiple <structure>-based templates possible.

Limitations of UIML instance documents. The current UIML template facilities are not a suitable solution for UIPs, since a strong tool support should define an instantiation configuration at design-time to raise efficiency and not the UIML document itself. With UIML as the basic configuration document there would be no overview about required parameters and no checking of constraints, e.g., the minimum, maximum or optional occurrence of elements), as there is even no definition of them inside the UIML document. UIML offers no visual aids in defining a UIP-instance. To conclude, reuse would still be limited to certain portions and GUI specification as well as configuration would pose high efforts.

Moreover, the above discussed strategies for applying UIML templates have another considerable drawback. The <d-template-parameters> definitions only allow for flat parameter structures. According to the presented examples, only the number of occurrences of child elements can be specified in the template and thus, configured in the UIML instance document. We cannot think of a way how to configure <style> information such as the label names for the given UIPs.

Summary. To draw a conclusion, UIML offers rich facilities like templates and restructuring mechanisms to manipulate a "virtual tree" structure [45] of a CUI model. Nevertheless, these capabilities are only valid for structure elements enumerated and defined concretely. There is no sufficient solution for the usage of a template, <repeat> or <restructure> for abstract elements with variable occurrences.

Currently, it seems that primitive UIPs may be defined via <part>-based templates, but the template has to be incorporated into a full UIML document and thus, variables have to be defined concretely. In addition, the limitations of parameter definition have to be taken into account.

In the following we provide a summary of current UIML shortcomings.

4) *Current UIML Limitations*

No meta-parameters for UIML documents. UIML provides no means to parameterize templates or UIML documents even further; meaning the introduction of meta-templates is not possible. UIML documents do not allow variables to govern nested templates. A higher level UIP configuration layer is missing, as indicated on the upper right hand side of in Figure 6. Such a layer could compensate for missing pattern support and allow nested parameterization for the final UIML document.

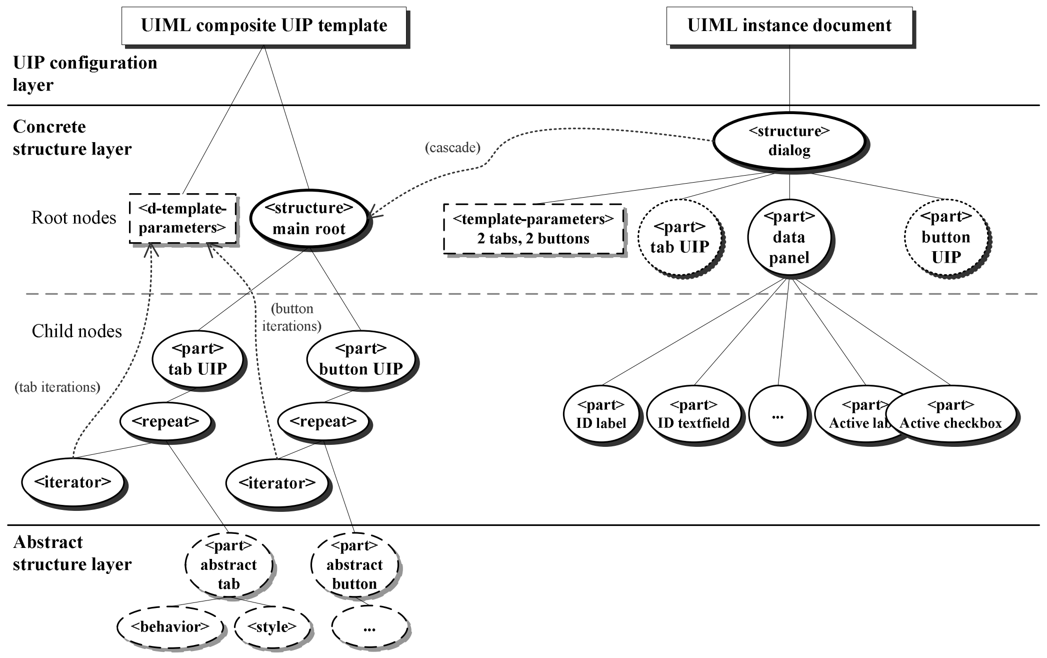

Figure 7. Schematic UIML <structure>-based template and its sourcing inside a UIML document

```
<!ELEMENT template (behavior| d-class| d-
component| constant| content| interface| logic|
part| layout| peers| presentation|
property| restructure| rule| script| structure|
style| variable| d-template-parameters)>
```

Figure 8. UIML 4.0 DTD [45] template tag definition

This way, the number of embedded template elements or respective sub-ordinate UIP instance could be governed. Currently, there is no reuse possible concerning root node structure elements with UIML, since the root elements are defined by the UIML UIP instance document itself. A developer would need to use UIML for defining final dialogs in detail this way.

Referencing abstract elements. Structure elements that are sourced from templates need to be referenced explicitly as this is needed for <style> and <behavior> sections for example. Therefore, a developer cannot specify the <behavior> or <style> of abstract elements or those yet to appear or being instantiated at design- or run-time inside a UIML document.

UIML DTD. Concerning the current UIML 4.0 XML DTD [45] as listed in Figure 8, the definition of templates may be faulty, since only one child element is currently allowed.

For instance, that means either <structure>, <part> or <d-template-parameters> are allowed as the solely child. Restrictions limit reuse to certain UIP combinations: Either one <structure>-based template in union or cascade as well as multiple <part>-based templates inside separately defined container elements are allowed. So a developer cannot specify how many template instances would be needed. Meta-parameters that would govern the individual template-specific parameters are not yet supported.

UIPs already instantiated. In the end, UIML itself is not capable of expressing complex UIPs. Only concrete template instances can be used, as they are configured concretely per <template-parameters> tag.

D. *Review of Content Criteria*

***UI-Control* types of UsiXML.** According to *UI-Controls,* UsiXML defines precisely which types of *UI-Controls* are available and what properties they can possess. An additional mapping model would have to be created in order to assign these elements to the entities of the target platform.

***UI-Control* types of UIML.** In comparison to UsiXML, UIML offers a more flexible definition of *UI-Controls*, since custom *UI-Controls* as well as their properties can be declared freely in the structure- or respective style-sections [45] without the need to define them beforehand. To map these structure parts to technical counterparts of the implementation, UIML offers a peer-section. This separate section can be used to specify a mapping between the parts defined within the structure and any target platform GUI component. The mapping to the GUI-framework can be altered afterwards without the need for changing the already defined UIPs. In addition, standard mappings can be defined and reused for a certain platform. However, the type safety like in UsiXML is not given. Thus, a homogenous usage of types and their pairing with properties has to be ensured by the developer and is not backed by the language specification like this is the case for UsiXML.

***Layout* definition of UsiXML.** Concerning *layout*, UsiXML uses special language elements to set up a GridBagLayout.

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

Behavior definition. Related to behavior, both languages define own constructs. Nevertheless, complex behavior is difficult to master without clear guidelines for both.

E. *Summary of XML GUI Specification Languages Review*

Besides the considered criteria for review, the two languages differ in indirect, supportive categories like framework and tool support or documentation. Additional comparison criteria and results of our evaluation are presented by TABLE I.

UsiXML and UIML may express structures similar to UIPs to some extent, but these resemble already instantiated patterns or their fragments. In fact, UIML may even express assorted UIPs through its template facilities. Nevertheless, these features are not sufficient for most UIP applications. In sum, both languages are missing the capability to specify UIPs properly.

F. *Valuation of model-based Processes*

Referring to the related work in Section III.C, promising solutions that enable higher reuse through the selection and instantiation of UIPs during specification and development of GUI systems are in reach. However, the presented approaches partly face the same challenges:

Common challenges. On the conceptual level, they need to review pattern relationships, enhance notations or probe the expression of more complex patterns or extend the set of supported patterns. For public evaluation, working examples of UIP instantiated to a certain context should be provided. Concerning tool support, researchers have to develop or enhance tools that aid in selection of appropriate patterns under consideration of possible relations among them. Moreover, tools are needed to guide the instantiation or configuration of selected patterns for a given context. Therefore, a solution finally adequate to fulfill each individual project's goals seems to be ahead of elaborate work in the future.

Common issues. In sum, we see some issues relevant to limit the effectiveness of further progress as follows.

Firstly, no detailed requirements or project goals have been communicated along with the presentation of concepts.

TABLE I. UIML AND USIXML IN COMPARISON

	UIML	UsiXML
language base	XML	XML
application	platform-independent user interface specification	device-, modality- and platform-independent user interface specification
reuse of code parts	by templates with assigned parameters	no
more than one user interface structure in one document	yes	no
manipulation of interface structures	through behavioral rules and replacement mechanisms of code parts	no, only method calls can be described
dynamic creation of interface structures	referenced through the use of variables	no, only static description
language documentation	extensive, with detailed language specification 4.0 [45] supplemented by descriptions and examples	2012: relative short, meta model described by class diagrams and short descriptions, no examples 03/2013: no updated meta model available
corresponding specification method and modeling framework	no, focused on implementation and prototyping	yes, implementation of Cameleon Reference Model (Task, Domain, AUI, CUI models), IDEALXML both as method and tool
tool support	GUI designer only	vast selection of tools (GUI designer, renderer, modeling framework, …)
rendering	XSL transformation, or compilation by own development	XSL transformation, rendering tools (XHTML, XUL, Java)

This hinders the evaluation of given approaches, and thus, their own justification and comparison to other approaches is hampered. More precisely, the UIPs defined as generative patterns and their capabilities remain a vague concept. Another considerable set-back is due to the fact that no detailed code examples or notation details have been presented yet.

Secondly, the general modeling framework and approach have been outlined as main assets, but no detailed architecture or transformations to code or final artifacts to be interpreted have been discussed so far. Up to now, the readiness of the approaches for practice or even their invented notations has to be questioned. For a more precise analysis of considered model-based processes reference [46] may be consulted.

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

Up to now, there have been no reports about experiences in the practical application of formal UIPs. The particular steps to be performed for a model-to-code-transformation and the shape as well as the outline of a formalization of UIPs are analyzed in the following sections.

A. Approach

To gain further insights about the practical implications of UIPs, they have been experimentally applied by two different prototypes. Similar to the probing of software patterns, selected UIPs were instantiated for simple example dialogs. These are illustrated in Figure 9.

Sketched examples. On the one hand, the examples consisted of a *view* fixed in shape that contained the UIP „Main Navigation“ [23] on the upper part. On the other hand, the lower part shows two variants for a *view* whose visuals are dependent on the input of the user.
Thereby, the UIP „Advanced Search“ [23] was applied. This UIP demands for a complex presentation control and is characteristic for E-commerce applications. Depending on the choice of the user, the *view* and *interactions* are altered. The search criteria can be changed, deleted and added as depicted in Figure 9 by two possible states. Both example dialogs should have been realized by formalized UIPs and one prototype.

Influences. Based on the current state-of-the-art concerning potential UIP notations, model-based processes employing generative patterns and the chosen example, we opted for two considerable different approaches and architectures.

Firstly, the potential GUI specification languages turned out not being capable of storing UIPs in a satisfactory manner. Only UIML was able to specify selected UIPs at design-time.

Secondly, the available sources of existing approaches provide no details about practical considerations and architectures related to UIP instantiation. In addition, they are affected by missing requirements for a definition and vagueness concerning the notation format of UIPs.

Lastly, the chosen dialog examples pointed out, that certain CUI models statically exists at specification time and others are due to change at runtime. Thus, a dynamic reconfiguration of a CUI model has to be considered.

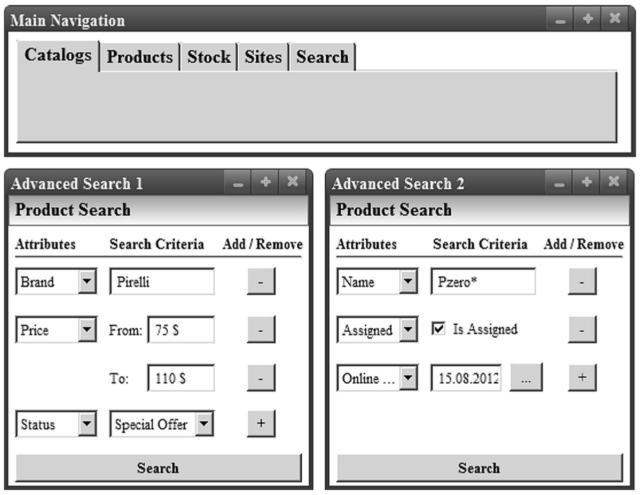

Figure 9. Example dialogs used for prototypes

Seissler et al. [36] also have outlined this aspect, but have not provided details yet. Finally, XML language capabilities will not be sufficient to provide proper formalization for dynamic user interfaces, as static user interfaces are already restricted.

Generation at design time. To test the formalization of simple UIPs and the generation of code for the examples, a solution, which generates the GUI dialogs at design time, was chosen. In general, the possibility to generate an executable GUI with the aid of UIPs had to be proven. The UIPs had to be completely defined at design time. Testing of the prototype had to be conducted after the GUI system was fully generated.

Choice of UIP notation. Regarding the structure of a GUI-specification, UsiXML proposes numerous models in order to separate the different information concerns *domain objects*, *tasks* and *user interface* as required by the underlying Cameleon Reference Framework. Not all the models were mandatory in terms of the example, since no explicit *essential model* was given. On the contrary, UIML operates with few sections within one XML-document. This is because the UIML format was easier to handle and learn with respect to the simple example. With UIML we could focus on the CUI to FUI transformation only.

In addition, on the basis of our review in Section VI UIML proved to be better suited for the specification of UIPs. Firstly, UIML is more compact in structure and enables a higher flexibility for shaping the formalization. Secondly, many of the language elements and models from UsiXML were not actually needed for the UIP „Main Navigation“. Thirdly, even the „Advanced Search“ example could not profit from the vast language range of UsiXML, since all possible variants for search criteria could not have been formalized or even enumerated. At least UIML offered the possibility to rely on templates in order to define all possible lines of search criteria composed of simple UIPs. UsiXML turned out to be too complex for these simple UIPs. Due to the limitations in documentation and the metamodel, it was not clear whether UsiXML permits the reuse of already specified UIPs at the time of our experiments. So we decided to apply UIML for the example dialogs.

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

In the following analysis, we mainly concentrate on the latter approach where the instantiation of UIPs is executed at runtime. In contrast, the generation at design time is an often applied variant with respect to available approaches outlined in Section III.C. This particular approach strongly relies on the employed formalization language for UIPs. In fact, this major asset is still challenged as seen in Section VI.F. Therefore, we can not provide further advances by practical application.

B. *Generation at Design Time*

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

Realization of „Main Navigation“. Java Swing was chosen as target platform. For the UIML <peer> section we decided to map the elements of „Main Navigation“ to horizontal JButtons instead of tabs.

In the formalization, the mandatory parameters for number, order and naming of *UI-Controls* were specified. As result, the UIP was described as an instance. The architecture was structured following the MVC-pattern [1]. The sections of UIML were assigned to components like this is illustrated by Figure 10.

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

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

Results. The prototype primarily was intended to prove feasibility. This is because we chose a simple architecture and did not incorporate a *Dialog Controller* for controlling the flow of dialogs. The control was restricted to the scope of the *UI-Controls* of the respective UIP. Thus, the behavior only covered simple actions like the deactivation of *UI-Controls* or changing the text of a label. Complex decisions during the interaction process like the further processing of input data and the navigation control amongst dialogs could not be implemented.

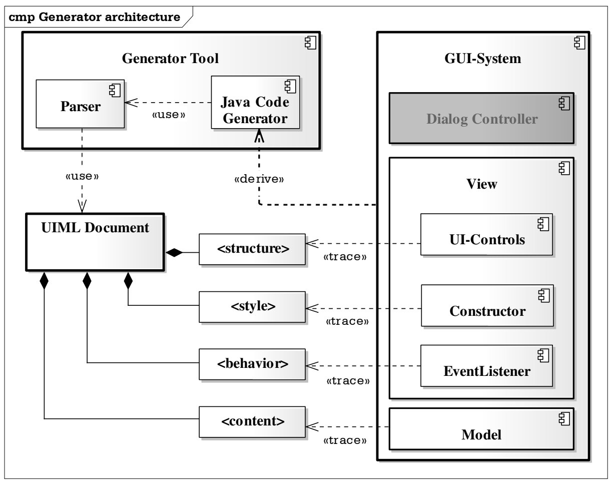

Figure 10. Architecture applied for code generation

A corresponding superordinate control could have been realized through a UIP-hierarchy in combination with appropriate guidelines for the formalization of control information. Despite the simplicity of the prototype, the following insights could be gathered:

Informal UIPs could be converted to formal UIP instances by using UIML as a formal language. Certain guidelines needed to be defined for this initial step. The *layout* of the example was specified by using containers for the main *window* and their properties. As a result, the *UI-Controls* were arranged according to these presets.

Nested containers and complex *layouts* have not yet been used for the experiment in this way. The <style> also was described concretely within the UIML document as well as the number and order of *UI-Controls*. The mapping of a formal UIP to a software pattern was described according to the scheme in Figure 10. Concerning the example *Advanced Search*, only fixed variants or a default choice of criteria could have been formalized. The generator could have created static GUIs accordingly without realizing the actual dynamics of this particular UIP.

C. *Generation at Runtime*

Since the *Advanced Search* UIP was very versatile and could not be formalized with all its variants with a single CUI model, the *layout* of the dialogs was fragmented.

By the means of a superordinate UIP the framing *layout* of the *view* was specified in a fixed manner at design time. In detail, the headline, labels and the three-column structure of the *view* appropriate to a table with the rows of search criteria were defined.

The mandatory but unknown parameters that determine the current choice of criteria and UIPs had to be processed at runtime. Accordingly, a software pattern had to be chosen that is able to instantiate UIP representations along with their behavior. This pattern had to act similarly to the builder design pattern [20], which enables the creation and configuration of complex aggregates. In [47] a suitable software pattern was discovered, which is explained shortly in the following paragraph and depicted in Figure 11:

Quasar VUI. The Virtual User Interface (VUI) is an early concept included in Quasar (quality software architecture) [48]. The VUI pattern follows the intention of programming dialogs in a generic way. This means that the dialog and its events are implemented via the technical independent, abstract interfaces *WidgetBuilder* and *EventListener* rather than using certain interfaces and objects of a GUI-framework directly. By means of this concept, the GUI-framework is interchangeable without affecting existing dialog implementations. Solely the component *Virtual User Interface (VUI)* depends on technological changes. Upon such changes, its interfaces would have to be re-implemented.

We are inclined that the VUI pattern implements some aspects symbolized by the CUI Cameleon step. Rather than specifying a certain CUI at design time and statically storing this as a source, the VUI creates a *Dialog* in an imperative way based on CUI level interface operation sequences.

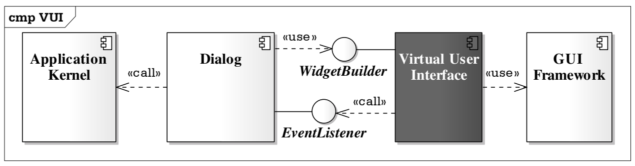

Figure 11. Virtual user interface architecture derived from [47]

By using the interface *WidgetBuilder*, a dialog dynamically can adapt its *view* at runtime. For instance, the *Dialog* delegates the *VUI* to create and configure a new *window* containing certain *UI-Controls*.

The *VUI* notifies the *Dialog* via the interface *EventListener* when events have been induced by *UI-Controls*. Both interfaces have to be standardized for a GUI system of a certain domain. This is essential to enable the reuse of reoccurring functionality such as the building of *views* and association of *UI-Controls* with events without regarding the certain technology or platform specifics being used. In short, an abstraction comparative to the CUI level and its advantages are enforced.

VUI for UIPs. The concept, the *VUI* is based on, can be adapted to the requirements of the UIP *Advanced Search*. The idea is to instantiate complete *view* components with UIP definitions besides simple *UI-Controls*. The *Dialog* is implemented by using generic interfaces, which enable the instantiation of UIPs, changing their *layout* and their association with events. In Figure 12 our refinement of the original *VUI* is presented.

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

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

Initially, events are reported to the *VUI* via *API-Events*. The *VUI* only forwards relevant events to the *Logical View*. When the respective event is solely related to properties of a *UI-Control* or a UIP instance, it is directly processed by the *Logical View* which delegates the *VUI* when necessary.

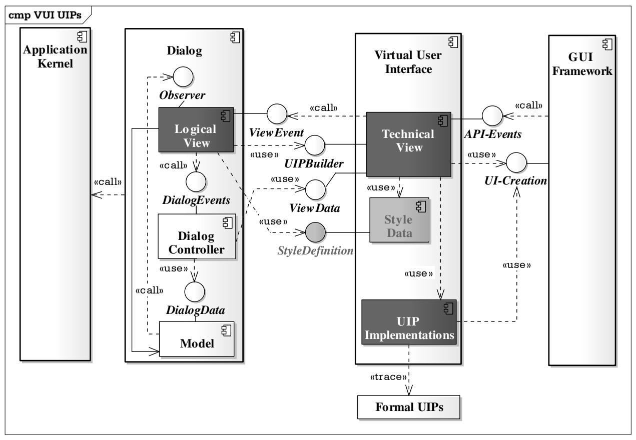

Figure 12. Virtual user interface architecture for UIPs

If the *Logical View* cannot process the particular event on its own, it will be forwarded to the *Dialog Controller*. For instance, this occurs when the user presses the button *Search* and a new *view* with the search results has to be loaded. The *Dialog Controller* collects the search criteria via the interface *ViewData* and sends an appropriate query to the *Application Kernel*. The result of the query will be stored as dialog data in the *Model*.

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

The UIPs had to be specified in a concrete manner like in Section VII.B. The prototype was not mature enough to handle abstract UIP specifications. The style of the *UI-Controls* was also described concretely, so the control of style by a component of the *VUI*, as depicted in Figure 12, has not yet been realized.

Through the *VUI,* the versatile combinations of *Advanced Search* could be realized according to the example at runtime. The VUI constitutes of a component-oriented structure related to the software categories of Quasar [48]. Accordingly, it possesses its virtues like the division of application and technology, separation of concerns therein and encapsulation by interfaces. Despite its challenging complexity, a flexible and maintainable architecture for dynamic GUI systems has been created. Finally, the formalized UIP fragments could be maintained at CUI level.

VIII. Practical Implications of User Interface Patterns

The reflection of both the theoretical implications of UIPs on GUI transformations and the results of our experiments led us to the following findings.

A. *Formalization of UIPs*

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

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

Parameters for UIPs. In order to overcome this static solution, a parameterization of formal UIPs has to be considered. Via parameters the number, order, ID, *layout* and style of *UI-Controls* within UIPs specifications have to be determined to provide a more flexible solution. Especially the number and order of *UI-Controls* have to be abstractly specified in the first place. In this way, UIPs can be applied in varying contexts. In place of a concrete declaration of style for each UIP, a global style template has to be kept in mind. By using this template, dialogs could be created with uniform visuals and deviations are avoided. For this purpose, the *VUI* incorporated the *Style Data* component. It is intended to configure the visuals of UIP instances and *UI-Controls* globally. The configuration is used for the instantiation of these entities by the *Technical View*. Consequently, style information from single UIP specifications could be avoided and the UIPs would receive a more universal format.

The model-based processes have already approached the formalization issues. In fact, they have detailed the parameterization of presented XML languages UsiXML and UIML for their custom modeling frameworks. However, we could not rely on their findings, as both detailed information was missing and considerable future work in the line of improvements was outlined. Yet, a more sophisticated solution has still to be invented. This conclusion is backed by our subsequent work to derive detailed requirements on the definition and application of generative UIPs [46].

B. *Generation at Design Time*

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

Transformation of abstract UIPs. Firstly, the UIP is abstractly specified along with all parameter declarations needed and placeholders for nested UIPs. Subsequently, these parameters have to be specified via a context model, which adapts the UIP to a certain application. Based on the abstract UIP specification and the context model, a model-to-model-transformation is performed in order to generate concrete UIP specifications like they were used in our examples. In this state, all required information is available for the generation of the GUI system. The described model-to-code-transformation can be performed as a follow-up step.

It has to be analyzed whether a suitable format is available to realize this approach, by means of UsiXML or IDEALXML and the respective AUI and CUI models.

C. *Generation at Runtime*

Regarding the UIP *Advanced Search*, it is clear that a large gap has to be bridged between the *essential model* and the *user interface*. A *use case*, which demands for such dynamic UIPs, hides a whole variety of different GUI-designs and thus CUI level models. Consequently, one static *user interface* cannot always be established for the elements of the *essential model*. However, even for these dynamic GUIs UIPs can serve as media to enable reuse of numerous aspects directly by generation along with a composition at runtime. The combined application of both our approaches can provide a feasible solution. Concerning the example from Figure 9, the previously generated *layouts* actually were reused for the *Advanced Search window* and the *views* of search criteria. By instantiation of matching UIPs, even the interactions respectively the presentation control was reused.

Generation of dialogs. As shown with our example, the current VUI is capable of the instantiation and composition of single parts of a certain *Logical View*. The generation of complete *Logical Views* on the basis of formal UIPs and their hierarchy could possibly be realized with the VUI architecture. The model describing the *Logical View* has to refer to the standardized interfaces of the VUI and a common UIP catalog.

To formally specify the UIPs to be used in this environment, only UIML currently seems to be suitable. Firstly, an analysis of the required and reused elementary UIPs as well as the relevant *UI-Controls* has to be conducted in order to populate the basic level in the hierarchy of UIPs. Next, these UIPs have to be formalized with UIML along with their required data types and invariant behavior that acts as a basis for presentation control within the *VUI*. Furthermore, the interaction and *layout* within the *Logical View* have to be specified using UIML as well. This is because UIML already offers templates that can be parameterized and thus used for the composition of several UIP-documents into one master document establishing a UIP of higher level. Concerning UsiXML, one dialog can only be specified by a single AUI or respective CUI model.

To complete the *Dialog*, meaning *Dialog Controller* and *Model*, relevant information on *tasks* and data *objects* has to be included into a formal model. The research on the collaboration between adaptable UIPs and these logical aspects already has advanced [6] [26] [29] [31].

D. *Limitations through the Application of UIPs*

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

Metamodels. The application of UIPs demands for clear guidelines for modeling of the *essential model*, which result in a second limitation. The rules for this model need to define stereotype element types and their delimitations. The definition of the *essential model* should be governed by a metamodel to ensure the uniformity of defined model instance elements. In this respect, it will be defined what types and refinements of *tasks*, *domain objects* and domain data types do exist in order to assign them homogenously to certain UIP categories. This concept is essential for the proposal of suitable UIPs for the automated development of GUI systems.

The proposing system needs to work in two ways: On the one hand, the GUI developer asks for a suitable selection of UIPs for a certain part of the *essential model* at design time. On the other hand, users need to be provided with suitable UIPs in dynamic dialogs at runtime based on their current inputs. The mechanisms can only work if a uniform *essential model* with clearly defined abstractions derived from fixed guidelines is available as fundamental information.

IX. Conclusion and Future Work

A. *Conclusion*

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

Currently, adaptability and reuse of UIPs are limited due to inadequate formalization options. Mostly invariant UIP and simple flat structures can be described by available template facilities of UIML. UIP compositions could only be created by manual implementation. We pointed to the limitations of current UIP specification format options and presented architectural solutions for their practical application. Above all, the upstream transformation of the abstract UIP description into UsiXML or UIML is worth to be considered, since one could use their strength in concretely specifying user interfaces. As an alternative to attempt to fully define UIPs in a single model, the approach to generate complete CUI level models on the basis of either UsiXML or UIML should be considered. Afterwards, the generation of GUIs based on this information would pose a minor issue.

B. *Future Work*

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

Paradigm. Another open issue exists in the field of interaction paradigms [12] and the applicability of UIPs. With respect to the procedural paradigm, processes are defined, which exactly define the single steps of a *use case* scenario. To provide a matching *user interface* for this case, additional information needs to be included in the formalization of UIPs. For instance, the process or *task* structures have to be specified by UIPs on a high level of hierarchy. These UIPs possess little visual content, maybe a framing *layout* for *windows*, and mainly act as entities for controlling the application flow. The *Dialog Controller* from Figure 10 and Figure 12 could be based on such a UIP. In this paper, no information for these components was integrated in the formal UIPs. So these components had to be implemented manually. For example, the *Dialog Controller* opens a new *window* with search results for the *Advanced Search,* controls the further navigation and delegates the structuring of the next or previous *windows*. In this context, our *VUI* solution is a compromise between automation and the reuse of elementary and invariant UIPs through manual configuration of the *Dialog Controller* and the delegated *Logical View*. A full automation needs further research and the consideration of the achievements other researchers have gathered so far in the field of task pattern modeling.

REFERENCES

[1] 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 2012), Xpert Publishing Services, July 2012, pp. 57-66.

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

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

[4] M. Haft and B. Olleck, "Komponentenbasierte Client-Architektur," Informatik Spektrum, vol. 30(3), June 2007, pp. 143-158, doi: 10.1007/s00287-007-0153-9.

[5] J. Vanderdonckt and F. M. Simarro, "Generative pattern-based Design of User Interfaces," Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS'10), ACM, June 2010, pp. 12-19, doi: 10.1145/1824749.1824753.

[6] 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 2007), Springer LNCS 4735, Sept. - Oct. 2007, pp. 256-270, doi: 10.1007/978-3-540-75209-7_18.

[7] P. Forbrig, A. Wolff, A. Dittmar, and D. Reichart, "Tool Support for an Evolutionary Design Process using XML and User-Interface Patterns," Proc. 5th Canadian University Software Engineering Conference (CUSEC 2006), CUSEC Proceedings, Jan. 2006, pp. 62-69.

[8] 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. New Trends (HCII 2009), Springer LNCS 5610, July 2009, pp. 826-835, doi: 10.1007/978-3-642-02574-7_92.

[9] 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. Design and Development Approaches (HCII 2011), Springer LNCS 6761, July 2011, pp. 299-308, doi: 10.1007/978-3-642-21602-2_33.

[10] M. van Welie, G. C. van der Veer, and A. Eliëns, "Patterns as Tools for User Interface Design," in Tools for Working with Guidelines, J. Vanderdonckt and C. Farenc, Eds. London: Springer, 2001, pp. 313-324, doi: 10.1007/978-1-4471-0279-3_30.

[11] M. J. Mahemoff and L. J. Johnston, "Pattern languages for usability: an investigation of alternative approaches," Proc. 3rd Asian Pacific Computer and Human Interaction (APCHI 1998), IEEE Computer Society, July 1998, pp. 25-30, doi: 10.1109/APCHI.1998.704138.

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

[13] J. Vanderdonckt, "A MDA-Compliant Environment for Developing User Interfaces of Information Systems," Proc. 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005), Springer LNCS 3520, June 2005, pp. 16-31, doi: 10.1007/11431855_2.

[14] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for Multi-Target User Interfaces," Interacting with Computers, vol. 15(3), June 2003, pp. 289-308, doi: 10.1016/S0953-5438(03)00010-9.

[15] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "USIXML: A Language Supporting Multi-path Development of User Interfaces," in Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Revised Selected Papers, R. Bastide, P. A. Palanque, and J. Roth, Eds. Springer LNCS 3425, July 2004, pp. 200-220, doi: 10.1007/11431879_12.

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

[17] H. Balzert, "From OOA to GUIs: The JANUS system," Journal of Object-Oriented Programming, vol. 8(9), Feb. 1996, pp. 43-47.

[18] A. Dearden and J. Finlay, "Pattern Languages in HCI: A critical Review," Human-Computer Interaction, vol. 21(1), 2006, pp. 49-102, doi: 10.1207/s15327051hci2101_3.

[19] N. J. Nunes, "Representing User-Interface Patterns in UML," Proc. 9th International Conference on Object-Oriented Information Systems (OOIS 2003), Springer LNCS 2817, Sept. 2003, pp. 142-151, doi: 10.1007/978-3-540-45242-3_14.

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

[21] 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)," Report of the Workshop Perspectives on HCI Patterns: Concepts and Tools, 2003 Conference on Human Factors in Computing Systems (CHI 2003), April 2003, http://www.cs.kent.ac.uk/people/staff/saf/patterns/CHI2003WorkshopReport.doc, 15.06.2013

[22] S. Fincher, PLML: Pattern Language Markup Language, http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html, 15.06.2013.

[23] M. van Welie, A pattern library for interaction design, http://www.welie.com, 15.06.2013.

[24] Open UI Pattern Library, http://www.patternry.com, 15.06.2013.

[25] A. Toxboe, User Interface Design Pattern Library, http://www.ui-patterns.com, 15.06.2013.

[26] F. Radeke and P. Forbrig, "Patterns in Task-based Modeling of User Interfaces," Proc. 6th International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2007), Springer LNCS 4849, Nov. 2007, pp. 184-197, doi: 10.1007/978-3-540-77222-4_15.

[27] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML: An Appliance-Independent XML User Interface Language," Computer Networks, vol. 31(11-16), May 1999, pp. 1695-1708, doi: 10.1016/S1389-1286(99)00044-4.

[28] X. Zhao and Y. Zou, "A Framework for Incorporating Usability into Model Transformations," Proc. MoDELS 2007 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2007), CEUR Workshop Proceedings, vol. 297, Oct. 2007, http://ceur-ws.org/Vol-297/paper8.pdf.

[29] 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 2006), Springer LNCS 4385, Oct. 2006, pp. 82-96, doi: 10.1007/978-3-540-70816-2_7.

[30] P. Forbrig and A. Wolff, "Different Kinds of Pattern Support for Interactive Systems," Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS'10), ACM, June 2012, pp. 36-39, doi: 10.1145/1824749.1824758.

[31] A. Wolff and P. Forbrig, "Deriving User Interfaces from Task Models," Proc. Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2009), CEUR Workshop Proceedings, vol. 439, Feb. 2009, http://ceur-ws.org/Vol-439/paper8.pdf.

[32] C. Märtin and A. Roski, "Structurally Supported Design of HCI Pattern Languages," Proc. 12th International Conference on Human-Computer Interaction. Interaction Design and Usability (HCII 2007), Springer LNCS 4550, July 2007, pp. 1159-1167, doi: 10.1007/978-3-540-73105-4_126.

[33] J. Engel, C. Märtin, and P. Forbrig, "Tool-support for Pattern-based Generation of User Interfaces," Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS'10), ACM, June 2012, pp. 24-27, doi: 10.1145/1824749.1824755.

[34] 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 2012), Xpert Publishing Services, July 2012, pp. 36-44.

[35] J. Engel, C. Märtin, and P. Forbrig, "HCI Patterns as a Means to Transform Interactive User Interfaces to Diverse Contexts of Use," Proc. 14th International Conference on Human-Computer Interaction. Design and Development Approaches (HCII 2011), Springer LNCS 6761, July 2011, pp. 204-213, doi: 10.1007/978-3-642-21602-2_23.

[36] 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 2011), Springer LNCS 6884, Sept. 2011, pp. 136-145, doi: 10.1007/978-3-642-23866-6_15.

[37] 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 2012), Xpert Publishing Services, Nov. 2012, pp. 296-303.

[38] C. Pribeanu and J. Vanderdonckt, "A Transformational Approach for Pattern-Based Design of User Interfaces," Proc. 4th International Conference on Autonomic and Autonomous Systems (ICAS 2008), IEEE Computer Society, March 2008, pp. 47-54, doi: 10.1109/ICAS.2008.36.

[39] L. Constantine, "The Emperor Has No Clothes: Naked Objects Meet the Interface", http://www.foruse.com/articles, 15.06.2013.

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

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

[42] UsiXML, abstract user interface (AUI) metamodel, http://www.usixml.org/fr/downloads.html?IDC=348, 15.06.2013.

[43] UsiXML.eu, http://www.usixml.eu/w3c, 11.03.2013.

[44] UsiXML, metamodels submitted to W3C, http://www.w3.org/wiki/images/5/5d/UsiXML_submission_to_W3C.pdf, 15.06.2013.

[45] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml, 15.06.2013.

[46] 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 2013), Xpert Publishing Services, May 2013, pp. 34-43.

[47] E. Denert and J. Siedersleben, "Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur," Informatik Spektrum, vol. 23(4), Aug. 2000, pp. 247-257, doi: 10.1007/s002870000110.

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