An Analysis of the Generative User Interface Pattern Structure

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

Abstract — **Current business information systems extensively rely on graphical user interfaces (GUIs). These sub-systems enable the interaction between the end user and application kernel services that are essential for the business process instances. Due to dynamic and rapid changes of both business processes and their required services, a strong need for the quick adaptation of GUIs to the occurring changes arose. As both efficiency and usability are essential for the GUI adaptation, model-based development processes that involve patterns and their instantiation for specific GUI contexts have been suggested by ongoing research. Being based on human computer interaction patterns, the new kind of pattern needs to be formalized in order to enable the automated processing of configurable instances by generator tools. However, current research is still at the edge to express the concepts for such generative user interface patterns. Crucial factors and impacts of those patterns have not been described sufficiently yet so that a standardized format for the expression of variability is still missing. With our work, we briefly review the current state on modeling user interface patterns and their requirement aspects. The ultimate objective of this paper is the development of an analysis model that is able to express both the structure and variability concerns of user interface patterns in detail. To evaluate and illustrate the analysis model concepts, selected user interface pattern instances are modeled via object models. As result, a detailed description of generative user interface patterns is achieved, which can be applied as a basis for the verification of recent approaches of model- and pattern-based GUI development or even the synthesis of a dedicated user interface pattern language.**

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

I. INTRODUCTION

A. Motivation

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

GUI requirements. In this context, a user interface primarily is required to fulfill both the criteria of functionality and usability. On the one hand, a GUI has to reflect the current process definition, and thus, offer access to the respective activities in order to provide effective support for the user. On the other hand, for this support to be efficient, the non-functional requirement of usability, which embraces the suitability for the task and learning, as well as a high degree of self descriptiveness [2], plays an important role for testing and the acceptance for productive runs.

GUI adaptability. As business processes tend to change over time, the functional requirements based on them, such as use cases or task models, may change considerably, too. With those changes taking place, new requirements, having a significant impact on the GUI artifacts, are being introduced. Consequently, this part of the system has to conform to a high demand on adaptability besides the first release-specific requirements. Especially standard software systems, which offer a configurable core of functions to support business models, like applied in e-commerce, see a distinctive demand for adaptive user interfaces [2][3]. Accordingly, a user interface of a business information system has to be based on a software architecture or development process, which facilitates the transition to new visual designs, dialogs, interaction designs and flows without causing significant costs in manpower and time.

Current limitations. Nowadays, the above mentioned requirements still cannot be accomplished fully by automation and generative development processes. On the one hand, available GUI-Generators can only cover certain stereotype parts of the user interface and may not lead to the desired quality in usability [3][4]. On the other hand, model-based development processes, which are able to generate more sophisticated user interfaces, also cannot support all variations on interaction and visual designs the changing business processes may demand for [5]. Finally, concepts that combine increased reuse and automation in user interface development and adaptation are being sought of.

User Interface Patterns. Together with other researchers [2][4][6][7][8][9][10][11], we believe that certain aspects of the GUI can be modeled independently in order to be composed and instantiated to their varying application

This is a revised and substantially augmented version of "An Analysis Model for Generative User Interface Patterns", which appeared in the Proceedings of The Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013) [1].

contexts. As evolution and individualism in GUI implementations generally induce high efforts, an approach has to be followed, which enables a higher degree of reuse, and hence, allows for more common basic parts to be shared among components. For reuse, the basic layout of a dialog, its positioning of child elements and navigation flow as well as reoccurring user interface controls (UI-Controls) and their data type processing are to be mentioned as candidates for automated generation. In this context, the occurring variability needs to be expressed by new artifacts in the development process chain. The need for a systematic description of reusable GUI artifacts arose and initially has found its expression in human computer interaction (HCI) [12][13][14] or, more recently, in user interface patterns (UIPs) [7][15]. In this regard, UIPs describe the common aspects of a GUI system in an abstract way. The developers concretize them with the required parameter information suited for the context of their instantiation.

UIP conception. The existing work about UIPs applied in model-based development processes [8][9][10] has laid down conceptual basics and milestones towards experimental proofing. However, no dedicated pattern definition for user interface development [6][16] has emerged yet, and so, the motivation of the PEICS 2010 workshop is still of high relevance [17].

Factor model. To progress towards a more detailed and complete UIP conception, we intensely elaborated requirements with impacts to architecture, formalization and configuration of UIPs in reference [5]. A process, which enables the instantiation of UIPs and their compositions to form a GUI of high usability and adaptability, altogether, needs such a clear basis of requirements. However, the factors we have modeled reside on a descriptive level that is not favorable to be directly translated to notations or formats for generative UIPs.

B. Objectives

The results of our work on the factor model in reference [5] have led us to the strategy to specify an analysis model for the UIP aspects and their various impacts. This model shall serve as a medium to close the gap between descriptive requirements of the factor model and formal notations. With the analysis model, we are detailing the requirements even more and progress towards a semi-formal notation for their description. The analysis model is intended to capture all essential aspects, properties and required parameters for context-specific application of UIPs. With this contribution, an initial version of the analysis model is presented.

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

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

C. Structure of the Paper

The following section provides an overview of the pattern type to be covered in this work. To begin with, origin and basic definition of UIPs are presented with the aid of related references from the human computer interaction community. To address possible formalizations of UIPs, XML based languages, which enable the platform-independent specification of GUIs, are introduced. In addition, an UML-based approach that promises formal modeling of UIPs on the basis of class models is briefly described as well.

In Section III, we present an overview of the role UIPs may assume with respect to the development of GUIs in the domain of business information systems. In addition, a UIP based development and modeling concept is briefly introduced to inspire a comprehensive view on UIPs. Lastly, requirements related to UIPs are reviewed to draw a distinction to common user interface development practices.

In Section IV, the problem statement is formulated. We summarize the outcomes of our previous work on the examination of model-based development processes and valuate the current state of related work.

The description of our approach follows in Section V. Our main achievement is the elaboration of the analysis model that is presented in Section VI. Object models that are presented in Section VII will reveal additional details of the analysis model applied to UIP examples. Therefore, the object models will evaluate the applicability of the analysis model. The results of our work are reflected in Section VIII, before we conclude and suggest future work in Section IX.

II. Related Work

A. Human Computer Interaction Patterns and User Interface Pattern Definition

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

HCI pattern ambitions. The early stages of patterns for user interfaces were determined by the goal to describe reoccurring problems and feasible solutions for GUI design offering high usability. Borchers [14] stated that human computer interaction (HCI) experts had a hard time communicating their feats in ensuring a good design of a system’s GUI to software engineers. Thus, the idea was born to express good usability via patterns as this was already a good practice for software architecture design. In this regard, Van Welie et al. [18] argued that patterns are more useful than guidelines for GUI design. In addition, they suggested the term pattern for user interface design along with criteria how to assess the impact on usability of each pattern.

Research into HCI patterns went on and culminated into pattern languages such as the one created by Tidwell [19]. Prior to this development, Mahemof and Johnston [12] outlined a hierarchy of patterns, what already implicated that there are complex relationships inside HCI pattern languages.

No unified pattern notation. Some years later, Hennipman et al. [20] claimed that available HCI pattern approaches could be improved as there were still obstacles for their efficient usage. Their analysis of relevant sources revealed major issues such as the missing guidelines how to formulate new HCI patterns, integrate them in tools and how to apply them. The request for a standard pattern specification template already was formulated in references [14] and [18]. In this regard, Borchers mentions early sources adopting the famous pattern notion by Christopher Alexander. Finally, Fincher introduced PLML [21] in reference [22]. However, the issue of a missing standardized pattern format still persists [17], which eventually is detailed by Engel et al. [23]. Therein, they analyze the shortcomings of current HCI pattern catalogs, the intended standard notation of PLML and its extensions.

UIP definition. Vanderdonckt and Simarro [24] separate two main representations of patterns based on the intended usage. Descriptive patterns serve a problem description and solution specification purpose. In contrast, generative patterns feature a machine readable format as they are to be processed by tools and in particular GUI generators. Besides this rather general segregation, we have not found any elaborate definition on UIPs.

B. Formal Languages for GUI Specification

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

In our previous work [3][15], we already went into the possibilities to express UIPs with the means of mature GUI specification languages UIML [25] and UsiXML [26]. As these languages are focused on platform-independent full-fledged GUI specification and intended to be machine processed, some of their elements may be candidates to be included in a sophisticated UIP definition model. Both languages feature common elements to define the visual layout, interactive behavior and content of a certain GUI part. For pattern-specific application, UIML and UsiXML differ in their capabilities: UIML incorporates elements for template definition and a peer section, which decouples structures or UI-Controls within the layout from their technical counterparts. In contrast, UsiXML is based on a more complex approach, which defines a metamodel consisting of a model hierarchy and methodology [27]. The abstract (AUIM) and concrete user interface model (CUIM) may be of relevance for our objective.

C. UML Class Based Modeling of User Interface Patterns

In our search for UIP aspects and definitions we discovered an approach towards UIP modeling that relies on UML. No exact UIP definition was provided either but on the basis of given examples the individual UIP aspects were outlined rather clearly.

The UML is a common basis for modeling software systems. As a notation it is present in major CASE tools and is applied to express multiple aspects and views of a system in one comprehensive model. To further complement the aspects of a system in this model, an approach for modeling UIPs with UML class models was developed by Beale and Bordbar [6].

Common motivation. Their motivation is sourced from several problems. Firstly, they support our claim from Section II.A that no standard specification for UIPs does exist. Secondly, available UIP catalogs or collections [19] [28][29][30] vary in structure as well as their pattern relations, so that developers would need considerable expertise to use those resources effectively or train new development team members. The problem stated by Beale and Bordbar is that no uniform principles for searching and identification of suitable patterns for a given context can be relied upon to raise effectiveness. This applies to the comparison of patterns between existing catalogs as well. Thus, pattern languages did not provide support for comparison between alternative patterns suitable for the same context and their trade-offs. In the end, the developer would be faced with a multitude of available options to select UIPs for the context or system in focus.

UML approach. As a solution for both problems, Beale and Bordbar follow the idea to express UIPs by the same means as used for the system model. In their approach, a tool reads a UML system model and suggests appropriate UIPs for GUI implementation or refinement. As input, the pattern matching tool analyses the system model's data structure provided as UML class model. Additionally, available UIPs are required as input models.

UIP representation. Each UIP is to be modeled statically as a class diagram, which incorporates both presentation and GUI data model elements with appropriate operations. With that representation "the behavioral and structural characteristics of an interaction artifact that provides a solution to an interface design problem" [6] is indented to be modeled. To complement the structure of a UIP, a UML sequence diagram is modeled that describes typical interaction sequences and may include stereotype functions like data loading and change of presentation states. For automation proposes, the sequence diagram can also be expressed via OCL.

UIP selection. During the processing, the UIPs are then matched to recognized structures within the system's class model. In the end, the developer is presented with all possible matching UIPs, which were found suitable for displaying the systems data structures. This may result in multiple choices, but the potential number of applicable UIPs for a given context is reduced to only matching structures.

Limitations. Beale and Bordbar do not claim to have found an ultimate solution. Their UIP representation is not intended to contain detailed pattern descriptions with forces, trade-offs and implementation hints like a full PLML specification would offer. In contrast, they limit their expressed UIPs to certain data structures and selected interaction elements with no user requirements.

Their primary goal was to analyze a system design model or a selected part of it in order to find proper UIPs to display the recognized data structures and to offer an ultimate selection of UIPs. No "aesthetic aspects" [6] or detailed visual design is captured with UIP models. To add these and more implementation related aspects, platform-specific models were suggested named "Device Profile Model" [6] to translate UIP models and generate specific instantiations.

They discuss another issue that stems from system model complexity and its variations, which depend on the skills and likings of the developers. Since a developer may model system design differently, the pattern recognition may produce different results. This is the same issue of varying detail of class diagrams where rather atomic units or composites may be chosen as model elements. Finally, these issues are to be treated by future work and in particular by an enhanced recognition algorithm.

However, the approach by Beale and Bordbar is closely bound to the data structure of a certain context or system. Therefore, the UIP definition is rather narrow and intended to fit within their set limitations. Following this approach, developers will soon seek for a more flexible UIP representation to fit the contexts of task and business process based systems. In addition, no implementation details were given for the data centered UIP concept.

III. Development of Business Information Systems with the Aid of User Interface Patterns

A. General Graphical User Interface System Development Artifacts

Before we look into the details of the UIP analysis, we would like to reflect the GUI development process and the potential role of UIPs therein.

General development steps. For each greater business information system the developers have to specify an essential model [31] that captures all necessary functional requirements. The artifacts of this specification are foremost kept independently from architectural and technical details. Therefore, the requirements usually contain no concepts for the GUI system. The transformation of requirements to a final user interface is no easy task to achieve [31]. Several modeling and refinement steps have to be undertaken where means for transformation rarely consist of automation tools. In reference [3], we already explored the theoretical implications of UIPs on these general transformation steps.

Artifact hierarchy. To reflect the role and value of UIPs in these particular development steps, we look closer at the involved requirement artifacts that are displayed on the left hand side of Figure 1. This figure and the following explanations will be used to argue that UIPs may be classified by several types, which reside on considerably different levels in a hierarchy. This UIP structure can be organized in parallel to the architecture artifacts in the middle column of Figure 1. Consequently, the matching UIP types are arranged on the right hand side of Figure 1.

Nowadays, requirements of business information systems are to be structured in a hierarchy of modular artifact types. This is due to the increasing complexity and number of requirements to be implemented. Requirements of higher level organize the structure and referencing of the lower level ones. Redundancies are avoided and concerns that form a modular structure are incorporated. These may lead the software architecture design and help identifying system related or implementation artifacts. For comparable reasons, UIPs should be organized in a similar fashion.

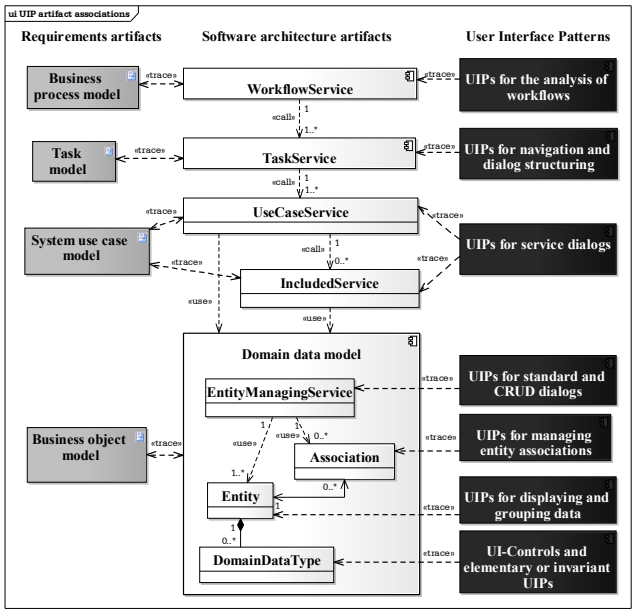

Figure 1. User interface patterns and software architecutre artifact relationships.

They should follow the architecture design levels derived from the requirements structure in order to provide a collection that is modular and reusable without redundancies.

UIPs related to development artifacts. Beginning at the highest level of specification, *business processes* are to be defined as requirements that guide the flow of events and *tasks* from the business goals perspective. They combine the system's as well as the company's resources logically and chronologically in order to realize certain business goals [32]. The part of their specification that is considering the system will be realized via *workflows* and their *services*. The *workflow service* practically is a technical implementation of the IT-supported portions of the *business process*. During its lifecycle it will interact with several applications at once in order to call the individual systems and their GUI implementation that offer access for the user to the realization of requirements situated in lower hierarchy. That is why UIPs will have to be considered mostly for these artifacts. Concerning the workflow itself, there may be UIPs relevant that enable the editing, monitoring and analysis of stored and currently running processes.

The next requirements level in the hierarchy is made up of *tasks*. One can argue whether *tasks* may be settled higher or lower in hierarchy than *use cases*. But that is not our concern at the moment. In Figure 1, *tasks* represent a manual activity of a *business process* as it is perceived by a single user or role. The *task model* captures structure and flow of functions or *use cases* that are combined to achieve the goal of the respective *business process activity*. Thus, the model arranges selected *use cases* to form a flow of events for a certain purpose. As these artifacts are mostly flow oriented, UIPs will be applied here, which determine the navigation and structure of dialog units the user needs to follow. This need was already investigated in reference [33] and acclaimed by other researchers [2][8][9][10] who incorporated respective task patterns.

Situated right beneath the *tasks*, *use cases* (actually system use cases) describe the interaction between user and system on a more detailed level incorporating references to the *domain data model*. In general, the user's goals and the systems provided *services* are specified in this respect. As many interaction steps, events and much data handling may be involved, the UIP type that complements these requirements will provide templates for dialogs that may be adapted to the individual context. Nearly the same applies for *included services* since they are shared among different *use case services*. UIPs may suggest sub-dialog types or portions of them for this type of shared service.

The last level in requirements hierarchy is represented by the *business object model*. The *business objects*, their relationships and data types relevant for higher level requirements are specified herein. Concerning the architecture, Evans [34] suggested an approach that merges analysis and design of respective artifacts into one coherent model, which uses similar stereotypes as depicted in Figure 1 as building blocks. These stereotype classes can be closely associated to certain UIPs. For instance, *entities*, which represent *business objects*, can be displayed by UIPs that arrange their data via tables, forms or other data views. Each time the *entity* is handled, the respective UIP may be instantiated and reused. This also applies to the *association types* the *entities* may use. Specialized dialogs that are applicable for editing certain object *associations* can be abstracted to UIP types. This principle can be followed for standard or CRUD (create, read, delete, update) dialogs, which are used solely for displaying and editing *entity* data. The UIPs only define the similarities of these common and reoccurring dialogs and adapt to the context by parameters like the concrete *entity* or *association* when instantiated. As far as the *DomainDataType* is concerned, there may be only certain UIP types needed for the objects that require a specialized view with a number of interaction options like calendars.

In sum, UIPs may work on different levels of abstraction and may be composed along this hierarchy of their associated artifacts. The requirements and their realizing architecture artifacts use a certain abstraction and structure for good reasons like handling of complexity and avoiding redundancy, so the UIPs should follow a similar structuring for consistent assignment. Finally, the scope for reusable UIPs is vast for business information systems since they should support a set of different architecture artifacts as this is drafted by Figure 1.

User interface development steps. Besides the structuring and assignment to their complementary artifacts, employing UIPs for GUI design involves some more development tasks.

Depending on the level of the considered architecture artifact, several UIPs must be brought together to form the user interface. In this regard, the developer has to arrange for dialog layout, choice and number of UIP instances, UIP instance positioning, and events as well as individual UIP instance visual states definition. Concerning the choice of UIPs, a developer may use the support of any suggestion tool and follow the principle that was presented in reference [6]. Furthermore, the developer needs to integrate the instantiated UIPs with the application kernel and its services. To do so, the UIPs should be able to be configured via parameters for data and action-binding. The former will be required beginning at the lowest level in artifact hierarchy when *DomainDataTypes* are to be bound to single UI-Controls or those contained within UIP instances. With respect to *service* artifacts, UIP configuration must facilitate the binding to actions that trigger the further processing and control by *services* discovered on top of the *domain data model*.

To conclude, there are various structures and related information on each stage to be considered when employing UIPs as reusable pattern artifacts.

B. User Interface Pattern Development and Modeling Concept

In this section, we briefly introduce the general considerations that seem necessary to approach an ideal UIP concept that can be employed in an artifact structure like illustrated by Figure 1.

Domain analysis. A development team may first start with an analysis what parts of the GUI systems are likely to be reused. They can consult existing descriptive UIP libraries like [28][29][30] to gather inspiration for future GUI visual specification. The selection of UIPs may depend on the domain and hierarchy of requirement artifacts.

UIP requirements model. The next step consists of the description of UIP capabilities. Due to the missing general definition of UIPs, there is no consent what are the actual requirements or features that UIPs must fulfill. In the previous section, we argued that UIPs should be sub-divided among several types that reside on a level in hierarchy that matches certain architecture artifacts. The UIPs have to feature properties that allow developers to customize their instances for corresponding artifacts. In addition, reusability and variability have to be specified in detail to enhance configuration facilities. Since UIPs will serve as abstractions for certain parts of the GUI system, they need to enable the same responsibilities with their specification. For all these concerns, an UIP requirements model should be established that fits the intended domain and grade of reuse. In the following section, we will present such a description model for UIPs that has been developed in our previous work.

UIP analysis model. When the requirements or features of UIPs have been pointed out clearly, the development team has to think about what structures, properties and relationships can be derived from the UIP requirements model. The task at hand is about the transformation of those requirements into detailed structures that prepare an information model, which will guide the later formalization of UIPs. This model primarily serves the purpose of a requirements analysis and is not intended for realization. The entities and their relationships derived from the UIP requirements can be modeled via a traditional object-oriented analysis model. As result, the analysis model should express all elements, properties, structures and relationships that will be needed by a language that will be employed to formalize UIPs for automation.

UIP meta model and formalization. On the basis of the analysis model, a formalization concept can be sought of. At this stage, the development team has to decide on the

abstraction level the UIP will reside on. More precisely, a decision has to be made how closely UIPs should be bound to target platforms and GUI frameworks. Vanderdonckt [27] presented the Cameleon Reference Framework, which can be consulted for further guidance. In this regard, the abstract user interface (AUI) level groups *tasks* into containers and their structure. Therein, UI-Controls and containers are only defined generically as abstract interaction objects (AIOs). These can be shaped very differently with respect to the next two steps: Concrete user interface (CUI) represents a common platform-independent basis model and final user interface level (FUI) embodies the device or platform model using the specific rendering units of the GUI framework.

We already analyzed this model in reference [3] and came to the conclusion that UIPs should be modeled on the CUI level. The CUI employs concrete interaction objects (CIOs) that refine the AIOs of the AUI. In detail, CIOs resemble a chosen set of both UI-Controls or containers and their respective properties sourced from common UI-toolkits or frameworks. To enable the platform-independent application of UIPs the CUI level should be chosen.

Finally, the developers have to decide on a language that facilitates CUI level modeling of UIPs. Depending on the analysis model, enhancements for existing languages may have to be developed. The parameters and variability concerns most likely need a new concept not already included in languages available in our days. In the end, a metamodel for UIPs has to be established that defines the logical elements being available for the formalization language. The refinement of the UIP metamodel may take several iterations as both analysis and requirements model may be changed several times and gain maturity. Moreover, mandatory and optional elements for UIP formalization have to be determined in order to prepare for different UIP types in the sense of a hierarchy symbolized by Figure 1.

Architecture artifacts metamodel. In parallel to the development of the UIP metamodel, the architecture artifacts of the domain have to be abstracted for forming a separate metamodel. This serves the purpose of mapping UIP types to matching artifact types. The specific artifacts and their stereotype properties have to be determined. The properties will be used to associate potential UIPs to architecture artifacts so that the developers will be presented with choices what UIPs will be generally applicable for a certain context. For instance, a date type as a *DomainDataType* of an *Entity* can be associated to a UIP consisting of a textfield and a connected date selector. Another option could be the presentation of a calendar UIP whenever this *DomainDataType* is encountered. Thus, both metamodels have to establish connections between architecture artifacts and UIPs since architecture properties will partly serve as parameters to enable action- and data-binding when configuring UIP instances.

Transformation concept. After the conceptual modeling has been completed, technical concepts for the transformation of instantiated UIPs into executable dialogs of the GUI system have to be developed. There are several options how to compose a solution. We are still considering these and only mention general directions since they are not in the scope of this work. Concerning principal architectures for generation, reference [3] can be consulted. In addition, there also is the possibility of using interpretation of CUI models. References [10] and [11] briefly described that approach for UIML.

C. Requirements Model for User Interface Patterns

Based on our previous work, we progressed towards an elaborate influence factor model for UIPs that is depicted in Figure 2. Motivated by missing standards and competing UIP notations inside modeling frameworks, this model was intended to establish an independent requirements view on the formalization and instantiation of generative UIPs: We took our examples and architecture experiments [3], as well as criteria, aspects and variability concerns [15], and refined them. The requirements stand close to the profile of current approaches in research. For details, reference [5] can be consulted.

As seen in the previous chapters, UIP and architecture artifacts should match. Thus, a UIP definition to be sought after has to introduce a pattern conception, which is backed by a limited set of types, roles, relationships and collaborations among GUI related specifications and components. Because of the complex nature of both GUI architectures and specifications, a restriction and specialization of the entities to be involved in the development environments for pattern-based GUIs have to be set. Along with this restraint, the GUI specific kind of pattern still needs to be abstract in order to enable vast customization and instantiation to differing contexts. The major share of the patterns vigor has to be sourced from the similarity in structural (*view aspect*) and behavioral (*interaction* and *control aspect*) definition of new GUI entities. In other words, the pattern definition introduces certain quality aspects in GUI design, which can be altered quantitatively, when they are respectively complemented with necessary structure, layout and style details (*view variability parameters*) as well as combined with each other (*behavioral* and *structural composition abilities*). This commonality ensures that no longer specialized solutions or manually refined structures, which cannot be covered by mere UIP instantiation, are applied in the same GUI system architecture.

Differences with UIPs. The question may be risen what will be the differences or benefits when taking the efforts to incorporate UIPs in the GUI development process compared to alternatives like GUI builders or CUI level based specification of a user interface with XML languages.

With UIPs, greater units of reuse will be employed as this is the case for CUI level languages and GUI builders. More precisely, complete dialogs or partly views of them can be configured as reusable units. Following the GUI builder or XML CUI specification approach, only small units situated on the UI-Control level or invariant views can be reused.

Along with the reuse of greater units, their interaction facilities and visual states may be reused as well. This kind of reuse is not possible with GUI builders or CUI models. Reuse would only be possible by copying and pasting large portions of existing CUI level code. Subsequently, the code has to be adapted manually to fit the changed context.

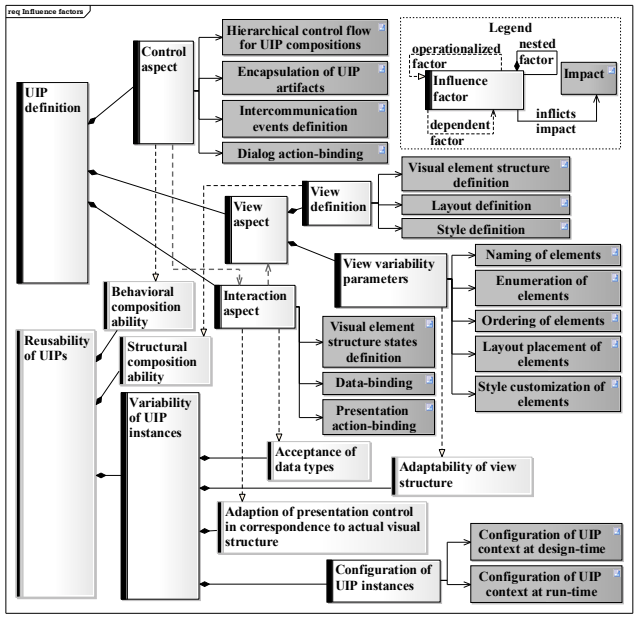

Figure 2. Influence factor model for generative UIPs described in reference [5].

By the application of UIPs, only the declaration of parameters that quantitatively alter the inner structure, states and behavior of defined UIP instances should be necessary in an ideal development environment. The quality aspect, and thus the general structure and behavior, should remain the same for all instances of the same kind of UIP.

Lastly, UIPs may enable their adaptation even at runtime when respective parameters have been specified [5].

Trade-Offs. The main issues while employing UIPs are the high efforts needed to establish a modeling concept or framework as outlined in the previous section. In addition, tools have to be developed that effectively support the developer in the formalization, selection and instantiation as well as rendering of UIPs. Moreover, most parts of the GUI architecture have to be prepared for automated processing with UIPs. In this current work, we only cover one single step of defining the general structure of UIPs via analysis.

IV. Review of Related Work and Problem Statement

A. UIP Definition

Descriptive UIPs. From our observations concerning descriptive UIPs, we learned that they are well-understood as specification elements and supported by the HCI community. Nevertheless, the research into descriptive HCI patterns has not yet converged towards a standardization for the structure and organization of UIPs [17][23].

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

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

As a consequence, there is a large gap concerning the detailed definition of generative UIPs. Thus, a format for UIPs has to be found that is at least able to express most impacts of *view* and *interaction aspect*. Filling the gap with their own UIP concepts and notations, the UML-based approach by Beale and Bordbar [6] as well as the recent model-based approaches will be analyzed in the following sections.

B. Modeling User Interface Patterns with UML

The approach by Beale and Bordbar introduced in Section II.C directly associates domain data structures to already modeled UIPs. This way a UIP can be derived from the context since the view structure (UIP model) is somewhat similar to the domain model or similarities can be identified thereupon.

Abstraction level. Referring to the Cameleon Reference Framework [27], the UML model of UIPs is situated at AUI level. There are no CUI or any specific visual details mentioned at all. Neither abstract nor concrete UI-Controls to be used for UIP elements are specified. Instead, a final user interface (FUI) level may be generated with the aid of the "Device Profile Model" [6]. It is not entirely clear to what extend the developers have to refine the existing UIP models for their instantiation.

No UIP metamodel. The modeled UIPs and their interaction sequences follow the UML metamodel facilities. There was no specialized UIP metamodel developed. In contrast, each UIP model represents a separate metamodel for certain instances to be created for the FUI. Therefore, they miss a generally applicable UIP description model, which governs adaptation or variability options. Those options are implicitly derived from the domain data model to be supported. The modeled UIP elements will adapt their child elements in correspondence to the attributes provided in the domain data model classes. Thus, the resulting FUI greatly depends on correct and complete modeling of the domain. In this regard, the "overview plus detail" (OPD) pattern might lead to false matches when "item" (C part) may not be detailed enough to justify a full "detail" view. This would depend on the actual "item" data structure and currently is not considered in the OPD UIP metamodel.

UIP factor support. A short comparison with our UIP requirements model reveals that certain aspects cannot be covered by the UML approach. Only *view* and *interaction* aspects are partly covered.

The *control* aspect or pattern composition is not directly considered at all. The structural UIP composition ability may be implicitly included when greater parts of a domain model or more classes with a number of relationships are analyzed. Then either multiple UIPs will be suggested to be applied together or a greater UIP metamodel has to be incorporated that matches the complete structure. Nevertheless, overlapping pattern definitions or composite UIPs that do

already employ smaller UIPs are not addressed with the required attention.

There are more restrictions concerning the *interaction* aspect. The partitioning or querying of data may not be prepared. In detail, the domain data must be displayed or processed on the GUI as modeled in the application kernel, so certain views or queries that may alter or merge data structures cannot be used for the original UIP selection. Otherwise the GUI data model must be specified separately and in detail so that the pattern recognition may finally work.

Many variability or parameter related impacts are to be derived implicitly from the domain. This applies for naming, ordering layout and style specification of UIP instance elements. Data- and action-binding may only be adapted in fixed limits of the defined sequence diagrams or OCL specification. The developers cannot configure non data-intensive aspects such as navigation, dialog structure and preparation of data inside views or dialogs and their level of detail with the reuse of available UIPs. Sometimes only selected attributes of an entity may be needed for display and not the entire attribute set of a domain class.

Benefits. Apart from these limitations, the aim of Beale and Bordbar primarily was to reduce the amount of UIPs to be taken into consideration for a certain *domain data model*. It may be beneficial when UIPs suitable for a certain *task* are to be suggested on the basis of a complex data structure available for analysis.

Supported artifacts. However, this may be a great restriction since the types of employable UIPs will be limited to certain levels inside the *domain data model* of Figure 1. So far, the UML approach only supports certain levels and special artifact relationships. UIPs are not subdivided concerning artifact support. In the next section, model-based approaches based on modeling frameworks mostly offer a more subtle classification of pattern types or their structures.

C. Summary of Model-Based Development Processes involving User Interface Patterns

The enhancement of model-based development by generative UIPs already found strong reception. In reference [5], we presented an overview and assessment of the approaches of Zhao et al. [2], PIM [35], UsiPXML [8], PaMGIS [9] and Seissler et al. [10]. For a summary, Table I compares these approaches.

In sum, the model-based approaches are converging concerning the *view* aspect, but ultimately failed to convey or inspire all UIP impacts. A summary of realized (arrow in a box) or inspired (single arrow) impacts is given by Figure 3.

Since our valuation revealed that there were many open issues associated with the different approaches, we only considered the full and no partly or probable realization of an impact. Notably is that the *view aspect* was realized by the most recent approaches. In contrast, the *interaction aspect* was only considered for *Data-binding*. Moreover, the *control aspect* was not realized by any approach, but inspired by PIM. Lastly, the *Configuration of UIP instances* was restricted to design-time only, but already inspired by Seissler et al. in reference [11].

TABLE I. COMPARISON OF APPROACHES FOR MODEL-BASED DEVELOPMENT EMPLOYING USER INTERFACE PATTERNS

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

Concerning the architecture artifacts, the approaches already incorporated pattern types dedicated to certain abstractions in the hierarchy of their modeling framework. Thus, the idea of matching UIPs and architectural artifacts or even patterns inspired by Figure 1 is already incorporated in those approaches to some extent. However, as they lack a clear requirements and structural definition of UIPs the mapping between artifacts cannot be considered as fully elaborated.

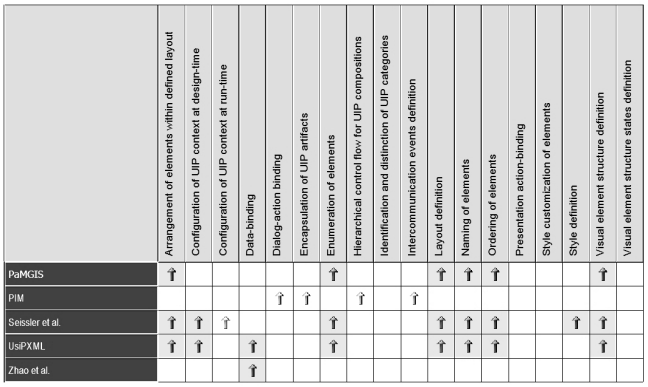

Figure 3. Impacts covered by examined approaches.

D. Formal GUI Languages and Model-Based Development

Enhancements. As there is still no dedicated language for UIP formalization, developers have to revert to existing GUI specification languages like UIML or UsiXML, which enable the specification of GUI parts on the CUI level. We will refer to them as XML languages in the following. As a result, two factions among the model-based approaches arose, one using UsiXML and the other applying UIML. Both languages need enhancements to express UIP related variability. Accordingly, the model-based approaches incorporated their own parameter and configuration concepts. In sum, they all failed to publish enhancements

that empower the specification languages regarding the *interaction* and *control* aspects. Currently, the notations are restricted to the *view* aspect mostly.

Generation of XML specifications. The XML languages have been developed to offer a platform-independent specification of GUI systems. In this context, they have been based on a metamodel that is somewhat similar to common universal object-oriented programming languages, which cannot handle aspects or traits and thus are incapable of expressing patterns with their abstract form. The XML languages clearly fail in the fulfillment of the reusability, variability and composition ability criteria [3][15].

However, applying the XML languages for their original purpose, apart from pattern definition, may play out their strengths. Accordingly, developers could use them for concrete GUI definition and final rendering to the desired platform. To integrate UIPs in this procedure, a generation of XML language code could be a possible solution to overcome the inabilities as proposed in reference [3]. This idea was already followed either by generation of UsiXML [8] or the interpretation of UIML [10]. The XML code would hold the already instantiated UIPs or the required information for rendering. The benefit would be the possibility to use existing tools for the XML languages. In addition, a more important merit would exist in obtaining a concrete user interface level (CUI) specification [27], and thus, the ability to be independent from platform specifics. In sum, the UIPs and their instantiation would be used to create CUI level models either based on UIML or UsiXML. The CUI model could be processed by the tools and transformed to target platform FUIs.

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

V. Our Approach

A. Strategy

As mentioned in the objectives, the impacts in reference [5] resulted in the strategy to develop an analysis model, which is aimed at further detailing the UIP aspects. We develop a structural model that is biased towards an implementation of a dedicated UIP language.

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

With a structural and more detailed model, the tracing of the influence factor impacts to potential solutions is better possible than with the pure influence factor model presented by Figure 2. In the factor model, there exist no separated entities that are modeled with their attributes and relationships to reflect a possible solution approach.

Assessment of recent approaches. Although we pointed out the factor support and issues we could so far discover as result of our assessment of other available approaches in reference [5], we also concluded that more details on examples and the applied notation have to be revealed in order to refine the assessment. By developing an analysis model, we seek to overcome the lack of detail and rationale on the design of notations suitable for UIPs. The notation to be used for modeling is the UML 2.0 class model.

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

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

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

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

solution aspects, a class model may provide a proper notation.

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

B. User Interface Pattern Examples

By reason that we do not want to claim being able to establish a UIP analysis model applicable for each domain, we stick to business information systems as mentioned in the introduction. More precisely, as stated in Section IV.A, we rely on common dialogs for e-commerce applications as a basis. In fact, we subsequently derive the analysis model by focusing both on the factor model in Figure 2 and the following example dialogs.

Simple search. For an easy example, we start with a dialog that has the "Search Box" [28] pattern instantiated. The simple search illustrated in Figure 4 is mainly composed by a single panel (*ContentPanel*), which defines a GridBagLayout as seen in the upper part of Figure 4. The UI-Controls are fixed and aligned in respective fashion. For variability, only the concrete object data types need to be bound to the combobox and textfield. In fact, this kind of UIP is mainly invariant.

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

Our example, depicted in Figure 5, mainly consists of two panels for layout definition as shown on the upper half. The panel *RootPanel* defines a GridBagLayout consisting of three cells (grey borders). Located in the center of this container, the *SearchCriteriaPanel* defines a layout of several rows each containing on cell (solid black borders). Additionally, the latter may grow or shrink in height to accommodate or discard search criteria lines to fit inside the container. Lastly, the *SearchCriterionPanel* (dashed borders) defines a layout appropriate for individual search criterions.

The usage of this dialog is as follows: Firstly, the user selects an object to be searched from the "Type of Object" combobox. Secondly, he chooses an attribute from the combobox inside the *SearchCriteriaPanel*.

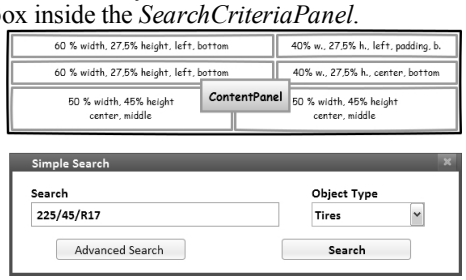

Figure 4. Simple search UIP example layout and dialog.

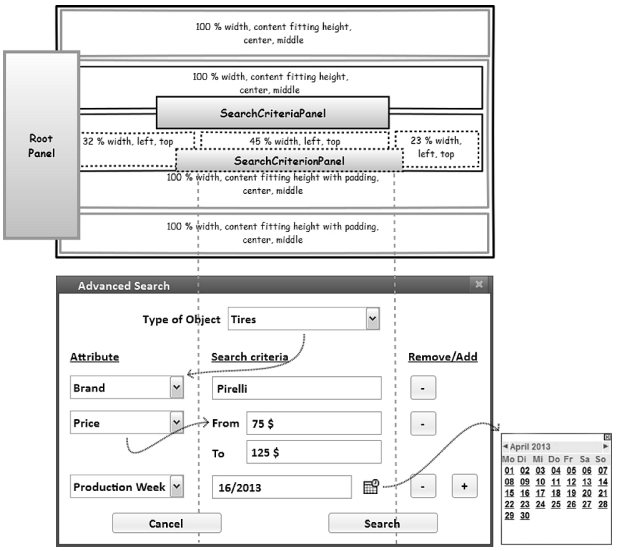

Figure 5. Advanced search UIP example layout and dialog.

Accordingly, the UIP dynamically has to instantiate new sub-UIPs, which resemble the single search criteria rows. For each datatype, a pre-defined UIP, which is similar in shape to the *SearchCriterionPanel*, is assumed to be available. In the example, the datatypes String, price, and week are considered. With the buttons on the right hand side, the user may add or drop new search criteria rows and so the view aspect will change. The variability is limited to the object types and their attributes to be searched with this UIP. Controller related aspects have to be adapted based on the UIP definition.

VI. The Analysis Model

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

A. Analysis Model Bias

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

Secondly, the analysis model may be biased towards requirements and thus acting as a traditional analysis model, which captures, refines and visualizes requirements. This option would be rather easy for the developers to understand, but would be costly to be translated to formal languages and generators. However, the translation to the XML languages is only a theoretical aspect, since generative UIPs cannot be

expressed by their facilities as discussed in Section IV.D. Eventually, we decided for the latter option.

B. General Rationale

Separation of definition and instances. A fundamental decision was the separation of elements or features that may be available in a UIP definition and the several element instances that may appear in a particular UIP application for a certain context. In other words, we divided the UIP analysis model into two parts. One part holds the definition and reoccurring elements (class names in black). The other part allows the description of instance information (class names in white) and individual element configurations. These basics are illustrated by Figure 6.

UIP configuration. Following the general concept of Figure 6, the main class *UserInterfacePattern* takes part in relationships that mostly focus on definition purposes, but also is connected to *UIPConfiguration*, which enables the description of particular UIP instances of the respective kind. The information used for pattern definition purposes will be covered in the following sub-sections. The configuration of UIP instances further branches into *Defaults* and *Parameters*. Both classes resemble containers that hold the *UIControl* instance information, which is declared as *UIControlConfigurations*, for a particular UIP instance. The *Defaults* are intended to omit stereotype configurations of default *UIControl* instances, which commonly appear in most contexts and shall not be re-defined redundantly.

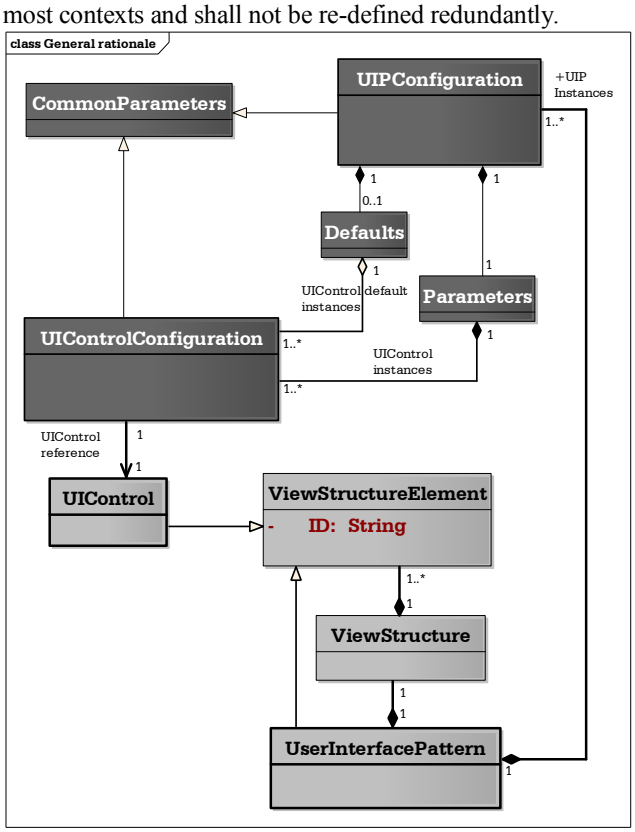

Figure 6. Gerneral rationale of the UIP analysis model.

Concerning the example dialogs, the basic or invariant *UIControls* needed for user understanding and interaction like the labels, textfield and combobox of the simple search should be defined as *Defaults*, as there is hardly any variability. This way, already established configurations may partly be reused among individual UIP instances. That means a UIP may contain pre-configured elements and parameters to avoid repetition. Later on, this facility will become useful for the dynamic adaptation of a UIP instance at run-time. Both *UIPConfiguration* and *UIControlConfiguration* are primarily used for the "Configuration at design-time" impact and thus contain the declarations a developer may define in interaction with an "instantiation wizard" [8] or any other configuration tool.

The configuration of *UserInterfacePatterns* and *UIControls* has to be separated, since both offer different sets of attributes, and more important, impact the GUI on different levels of abstraction or scope. This consideration also takes the possible artifact relationships of Figure 1 into account.

C. View Aspect Design

View definition. To begin with "View definition", this factor defines the *UIControls* or *UserInterfacePatterns* to be generally contained or allowed in a UIP specification unit as visual components. Both resemble a *ViewStructureElement*, which has a unique *ID* as identifier inside the pattern used by *UIPConfiguration* and *UIControlConfiguration* to reference the respective element. In this respect, *UIControl* is a classifier for the various visual components or widgets a GUI framework may possess as types. Figure 7 details the described relationships.

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

The style impact is not detailed here, since we have not came to a result in this regard and focused on the other impacts. For the sake of uniform views and maintaining corporate design, style information may be governed globally and locally by each individual *UIPConfiguration*. In addition, there may be constraints for each element, which determine its allowed minimum and maximum occurrences.

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

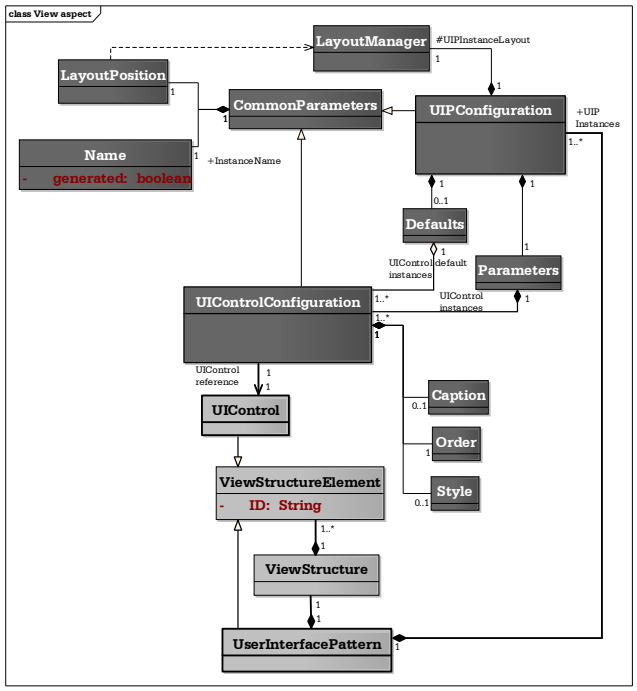

Figure 7. View aspect concepts of the UIP analysis model.

Referring to UsiPXML [8], layout patterns can be defined separately from presentation patterns. How they are integrated at various stages in the hierarchy, and more important, how they can be handled dynamically at run-time, remains an open issue, as there were no detailed examples for pattern composition and specification code given.

In addition, it is arguable whether a layout is assigned separately to a paralleled UIP composition or if each UIP models layout partly but explicitly. Partly means that UIPs need to define attributes for the number of rows and columns of a grid, their relative width and height, as well as the alignment. A visual impression of the abstract layout definition expressed by UIPs is depicted in the upper parts of Figure 4 and Figure 5. We decided to model this information by UIPs, as for advanced search, the layout needs to be re-configured dynamically with respect to *SearchCriteriaPanel*. This panel may grow and shrink in row numbers.

Layout definition. Inspired by our examples, we treat the layout container as a UIP, and thus, a layout pattern is already merged inside. So, the above mentioned layout definition parameters have to be associated to each *ID* of a UIP-type class, since it is acting as a superior container. Consequently, the advanced search dialog consists of three UIPs designated as containers in Figure 5. Translated to GUI frameworks, this implicates that each UIP will be treated as a panel or even window frame with a certain *LayoutManager* attached. We reason our approach with the fact that every dialog at some stage needs layout containers and these are eventually to be mapped to peers in the GUI framework. The detailed parameters for layout, such as padding, orientation and size policies, may be governed globally.

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

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

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

Both *UserInterfacePattern* and *UIControl* share some parameters defined as *CommonParameters*. For both *ViewStrucutreElements* the *Name* and *LayoutPosition* may be declared.

D. Basic Interaction Aspect Design

In the factor model of Figure 2, the *interaction aspect* was not separated between stereotype definitions and parameters, as this was done for *view aspect*. Finally, the main classes, which model the *interaction aspect*, all do resemble parameter types. Since the factors apart from the *view aspect* ones mostly embody cross-cutting concerns, the resulting *interaction* and *control impacts* refer to the static and variable declaration of *view impact* elements as a basis. This relationship is outlined with the dependency between *view* and *interaction aspect* in Figure 2. In detail, the *interaction* related *UIControlConfiguration* parameters comprise of *DataType*, *PresentationEvent* and *EventContext* as an additional child of the latter.

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

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

The latter point may resemble GUI architecture models discovered in common MVC architectures. The mentioned coupling points are either evaluated (events) or processed by the dialog kernel or logic part of the dialog. It is not necessary for that component to know where data changes and events have originated from. So, these suggested coupling points may be a good starting point. Accordingly, events (*PresentationEvents* and *OutputActions*) and the “GUI Data Model” have been included in the analysis model. These features originate from our thoughts about artifact relationships in Section III.A, and in particular, the association of *domain data model* elements and UIPs.

Data-binding. The binding of a *UIControl* to certain data is accomplished by a *UIControlConfiguration* parameter. So, the *DataType* binds the elements to certain data structures. As *DomainDataTypes* may significantly differ from the types used by the GUI framework, the class *GUIProjection* is rather associated as the configured *DataType*. For the *DataType*, it can be configured if the data is to be displayed only (input) or if the user may conduct changes (output), which are finally applied to the GUI *Model* part. The *DataType* parameter also may be associated to *EventContext*, which configures the data to be submitted by a *PresentationEvent* of the respective element. The diagram of Figure 8 details the *interaction aspect* data-binding considerations.

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

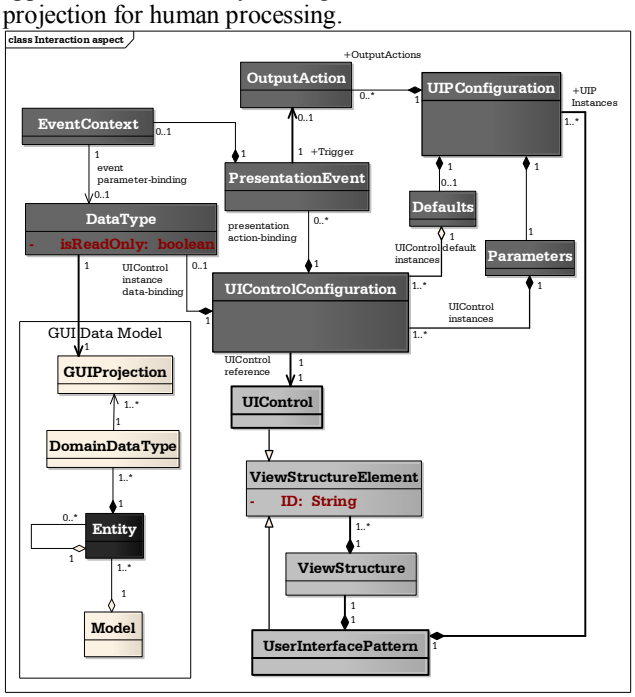

Figure 8. Interaction aspect data-binding concepts of the UIP analysis model.

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

Events rationale. For *PresentationEvents*, we enumerated some typical events implemented in GUI frameworks that may be triggered. To progress towards a unified solution for generative UIPs, we think that a standardization of events, *PresentationEvent* as well as *OutputAction*, and similar types is necessary. Figure 9 displays elements of the analysis model relevant for events.

The integrative and strict type definitions of the GUI specification language UsiXML on CUI level [27] may be a valuable resource for that approach. Otherwise, both specification and tool processing would demand for niche solutions that are hardly manageable with respect to versions and dependencies. We wonder how UsiPXML [8] or the UIML UIP definition by Seissler et al. [10] are defined as a language to be integrated in tool environments, which are to handle the generic concept of their variables and assignments effectively. We have to wait for them to publish detailed language definitions and code examples.

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

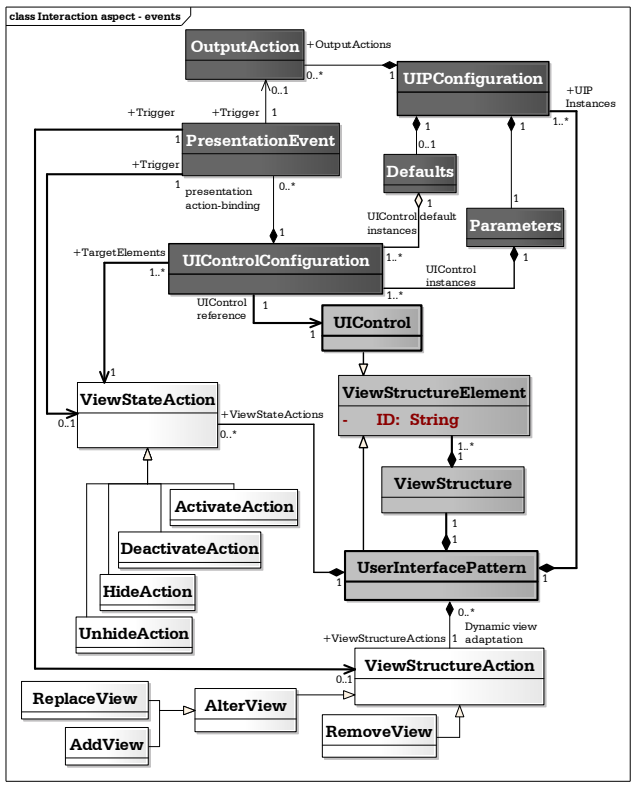

Figure 9. Interaction aspect event concepts of the UIP analysis model.

E. General Object Model View

To clarify the basic rationale the UIP analysis model is founded on, we will explain the general structure of an UIP artifact viewed from an object model's point of view. Figure 10 illustrates the basic objects to appear in each UIP specification. The structure of the analysis model leads to a hierarchical ordering of elements to be used for UIP specification.

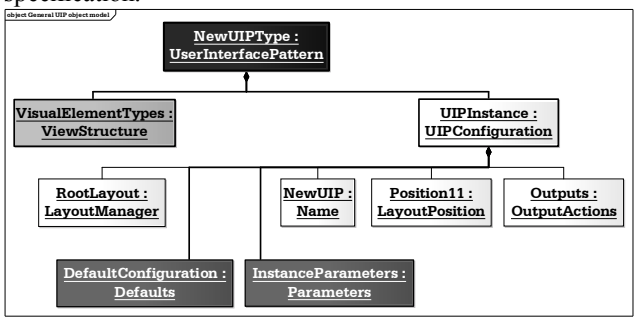

Figure 10. General structure of a UIP artifact based on the analysis model.

In this regard, the first two objects to appear are one *ViewStructure* for the definition of available *UIControls* or even nested *UserInterfacePatterns* and one *UIPConfiguration* that will be used to adapt the UIP instance object to the context. A *Name* and *LayoutPosition* with respect to the parent *LayoutManager* are to be specified as *CommonParameters*.

The parameters are shared among *UserInterfacePattern* and *UIControl* objects, so that both kinds of *ViewStrucutreElements* may be named and placed concerning layout.

One *UIPConfiguration* object with a reference to the main *UserInterfacePattern* object is mandatory. There may be more than one *UIPConfiguration* object when nested *UserInterfacePatterns* do occur within the *ViewStructure* object. With the respective *UIPInstance* object, all instances based on the available elements from the *ViewStructure* will be created. In addition, the general layout or *RootLayout* and the *OutputActions* relevant for architecture integration will be defined with that object, too.

The *UIPInstance* object holds two more configuration objects. On the one hand, a *Defaults* object will enable the reuse of the common configuration of that particular UIP. Therein, stereotype instances created from the available *UIControl* elements of the *ViewStructure* will be configured for the convenience of reuse. On the other hand, context-specific instances based on the *ViewStructure* specifc *UIControl* elements can be created with the *Parameters* object in parallel.

F. Advanced Interaction Aspect Design

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

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

Embedded UIP states. Since the possible states for composite UIPs cannot be enumerated or state machines finitely defined inside pattern specifications, we employ information, which describes the results of the state change, and thus, enables a generator to build appropriate state machines or comparative implementations. In Figure 11, relevant elements for the specification of dynamic UIP structures are displayed.

The *ViewStructureAction* is designed to handle the change of visual states for UIPs. For the trigger, a respective *UIControlConfiguration* is needed, which is aimed at a certain *ID* to allocate the *UIControl* and the type of *PresentationEvent*. We considered the addition, replacement, or removal of UIP instances. This behavior is closely related to the <restructure> tag of UIML [36] and may be refined based on its semantics. However, for UIML these facilities can only be applied with already instantiated UIPs.

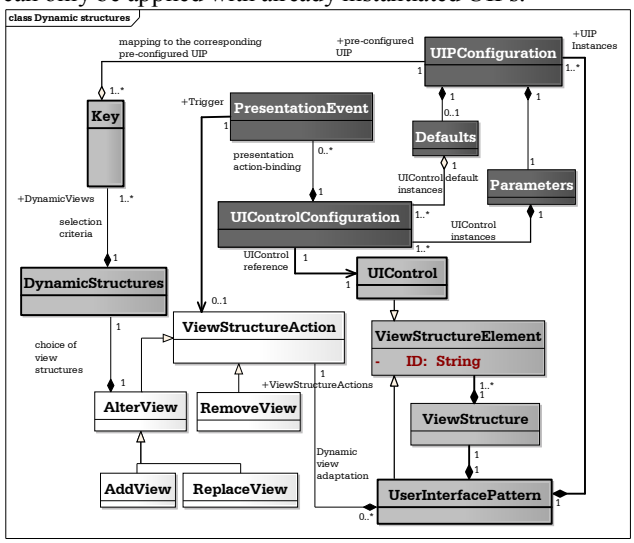

Figure 11. Interaction aspect embedded UIP states concepts of the UIP analysis model.

DynamicStructures are used for the addition, removal or replacement of *UserInterfacePattern* instances at runtime. They are selected on the basis of defined *Keys*, which enumerate certain *DataTypes* or *EventContext* data to assign pre-configured *UIPConfigurations* to the triggered *ViewStructureAction*. A *UIPConfiguration* may be used by more than one *Key*, which models a certain context situation. Concerning the advanced Search example, the *Model* holding the object and attributes lists must return values that

match the specified keys. Each time a combobox is changed, the presentation event handling routine must query the *Model* for the selected object's attribute and its kind or type of representation. The query result will be embedded in the *EventContext*, which is matched to a *Key* value. This way, the UIP and its *DynamicStructures* are based on a canonical representation of *DomainDataTypes*.

Moreover, the *ViewStructureActions* rely on pre-configured elements, which may only allow for variability concerning the *DataType*. They either rely on a self-reference (removal, replace) or additionally are associated to available elements of the *ViewStructure* (add, replace) via *DynamicStructures*. However, this mechanism only makes sense for *UserInterfacePatterns*, which are specified by *Defaults* and always represented by default *IDs* present inside the *ViewStructure* of a UIP definition. In this way, the *DynamicStructures* will only affect default or invariant *UserInterfacePatterns* inside the given *ViewStructure*, hence it is not desirable to replace entire sets of UIP instances defined on behalf of the developer for a specific context. Thus, manually defined UIPs portions have to be separated from *DynamicStructures*.

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

G. Control Aspect Design

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

H. Structure View on the Analysis Model

The resulting analysis model is illustrated by Figure 12. The classes shaded in medium grey are related to the "view definition" factor. Configuration related classes are shaded in dark grey and feature a white caption. Most *interaction aspect* impacts are supported by the classes shaded in white.

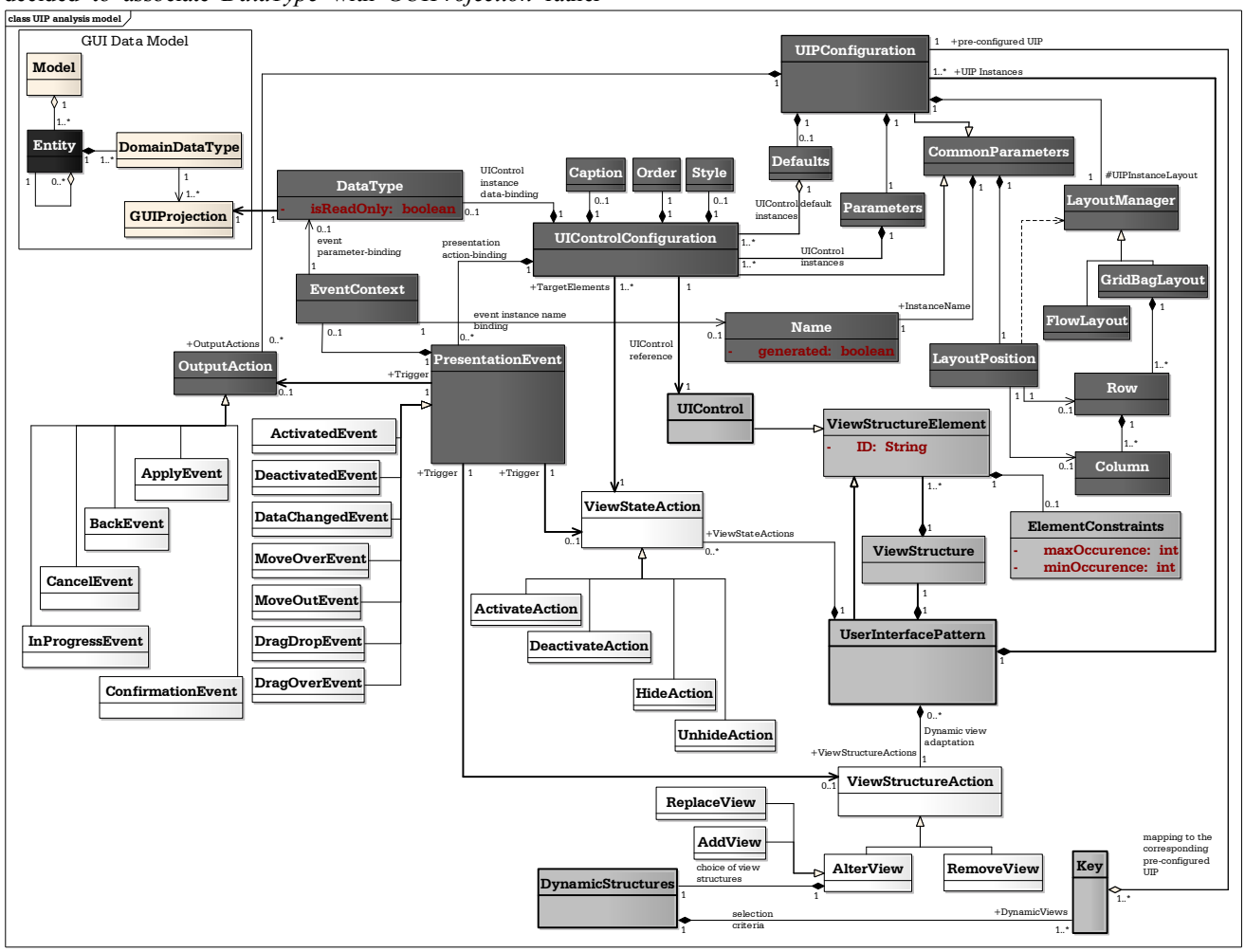

Figure 12. User interface pattern analysis model.

VII. Instance View on the User Interface Pattern Analysis Model

A. Simple Search Instance View

In this section, we apply the above described analysis model to the first UIP example entitled simple search. For that purpose, object models will be presented that are used to illustrate the different aspects of the UIP instance configuration. Please note that due to space limitations not all mandatory associations or objects will be modeled.

Since the simple search is mostly an invariant UIP, there is a need for a default configuration. Instance parameters will be limited to the *DataTypes* associated to the search input textfield and objects to be searched, which are determined by the user through the combobox listing available object types.

ViewStructure. To begin our analysis of that example, we enumerate the *UIControls* that are to appear as visual elements in the *ViewStructure* of the UIP. There are labels for designating the visual elements for the user, a texfield for search input, a combobox that holds object data and two buttons for triggering *OutputActions*. Each of these elements was incorporated into the *ViewStructure* on the left hand side of Figure 13. The label and buttons only appear once since their needed instances will be configured as *Defaults* holding *UIControlConfigurations* accordingly.

Defaults. The tree with *DefaultConfiguration* models the real UI-Controls that will appear on the screen when simple search is instantiated. For each *UIControl*, the caption and layout position are specified. Some labels have been skipped for presentation purposes; their configuration is performed in analogy to the other labels present in the object model. As far as the buttons are concerned, additional *PresentationEvents* are declared that are of the type *ActivatedEvent*.

UIControlConfiguration. Besides *Defaults*, the *UIPConfiguration* declares a *LayoutManger* used for positioning the *UIControl* instances. The two possible *OutputActions* are also specified on the same level.

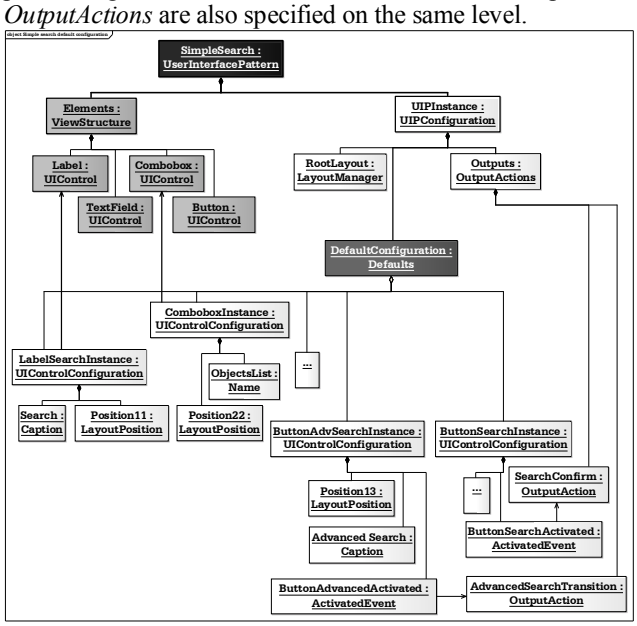

Figure 13. Simple search instance default configuration object model.

Lower in hierarchy, they are associated to triggering *PresentationEvents* that belong to certain *UIControl* instances, and more precisely, their *UIControlConfigurations*. This information is typical for that kind of UIP and can be reused by the *Defaults*.

Variability parameters. To adapt the UIP instance to the specific needs of the context, *Parameters* will be declared as depicted in Figure 14. The missing information for the data relevant *UIControls* is added here. To reference the same *UIControl* instance, the same *Name* has to be used during specification. For instance, the textfield and combobox are already present inside the *Defaults* object.

For both the texfield and combobox the data-binding is specified with reference to an existing *GUI data model* based on the *Entity SearchObjectData*. The latter will be processed by an application kernel service and has no GUI related responsibilities.

To increase the variability of that UIP, one could think about adapting the layout via parameters but this is currently not reflected in our analysis model.

The example is rather simple since the UIP has no visual states that cannot be handled implicitly by the facilities of the implementing GUI framework. The complete object model is provided with Figure 15.

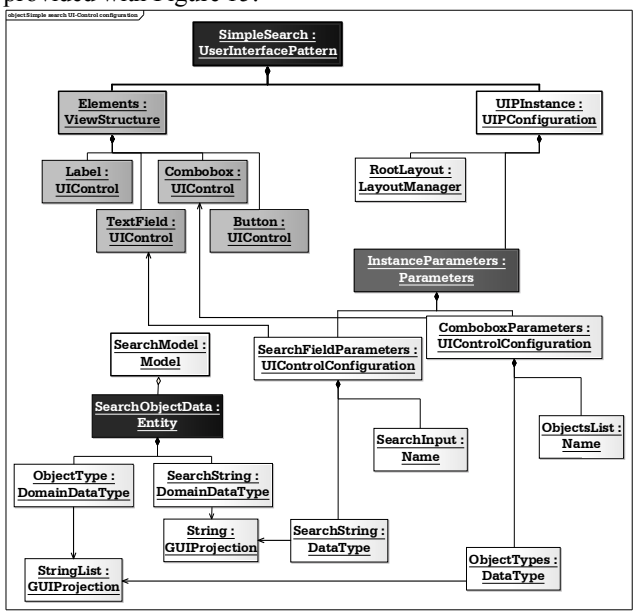

Figure 14. Simple search instance UI-Control configuration object model.

B. Advanced Search Instance View

The advanced search is far more complicated than the simple search object model. Therefore, we begin with a state chart that displays a set of a few possible alternations of the view state. In Figure 16, the state chart of the advanced search example is illustrated.

It is obvious that the visual element structure states are altered, each time the user performs a relevant input such as the selection of an object, an attribute or the activation of a button. The advanced search example involves a complex structure of nested UIP instances.

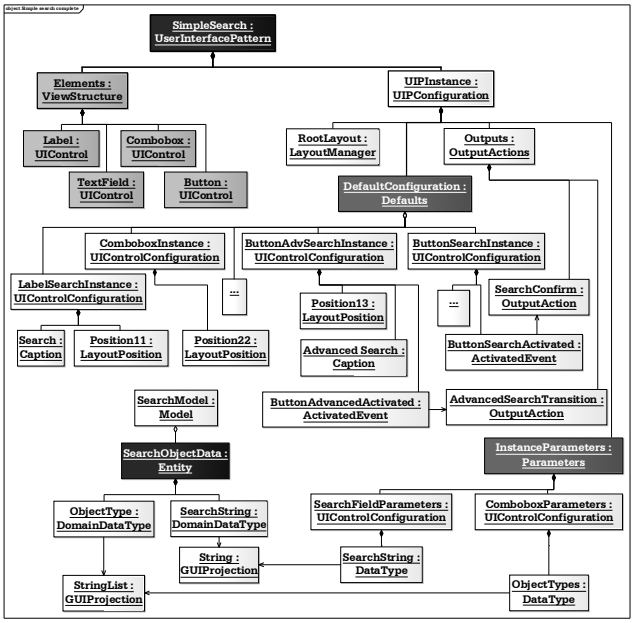

Figure 15. Simple search instance complete object model.

To begin with, the main instance will be one object of the advanced search UIP itself. Moreover, the pattern consists of a lower *ButtonBar* and a *SearchCriteriaPanel* that is built dynamically during user interaction. These basic UIP instance objects are arranged in the object diagram of Figure 17. The *MainInstance* holds configuration information about the data origin of the object types, so that the user may begin with a selection of the object to be searched. When the user has made his selection, a *DataChangedEvent* will be triggered as a consequence of the user interacting with the first combobox of the dialog and its respective *UIControlConfiguration*.

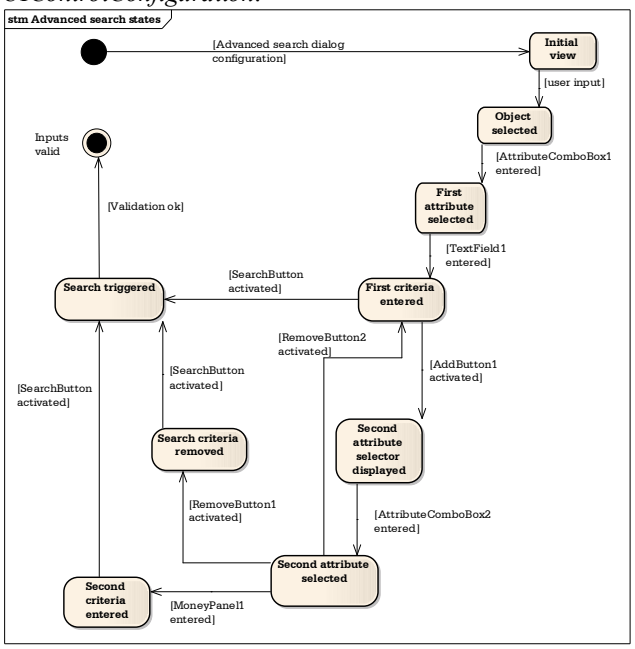

Figure 16. A few selected states of the advanced search example UIP.

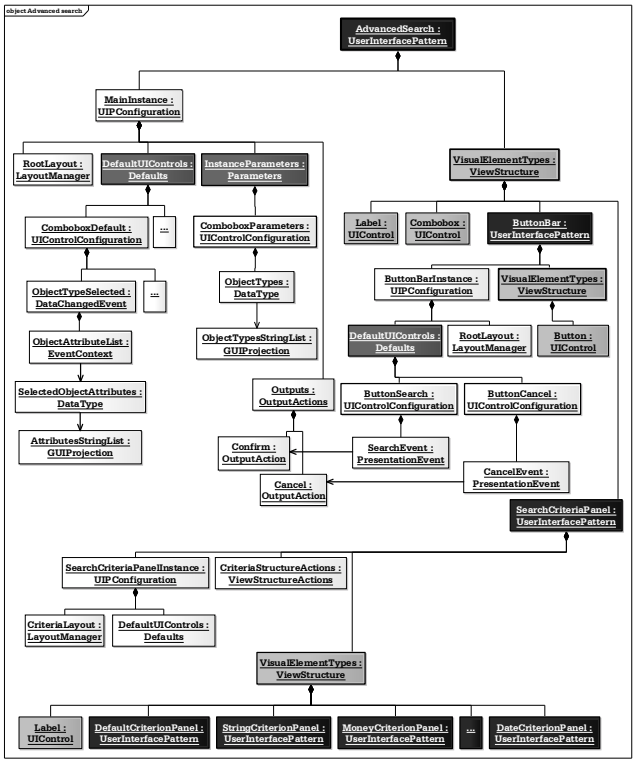

Figure 17. Advanced search basic object diagram.

The event will be associated with an *EventContext* to submit the attribute list of the selected object type to the lower situated search criteria comboboxes. The *SearchCriteriaPanel* constitutes of a number of embedded *UserInterfacePatterns*. These UIPs will serve as templates for the dynamic instantiation of search criterions. Notably is the *DefaultCriterionPanel*, which will be instantiated first when a selection has not been made by the user yet.

A detailed view on the *SerachCriteriaPanel* reveals the structure of the embedded *DefaultCriterionPanel* that is available in the *ViewStructure* of the former. The *DefaultCriterionPanel* defines a *ViewStructureAction* that allows for the replacement of the complete UIP instance with a pre-configured *UserInterfacePattern* of the parent *ViewStructure*. This replacement will be triggered when the attribute to be searched is entered by the user. A respective combobox *UIControlConfiguration* is present in the *DefaultCriterionPanel* default configuration. Depending on the attribute selected, an appropriate *UIPConfiguration* is determined via the evaluation of the *EventContext* and stored *Keys* of the *DynamicStructures*. Figure 18 provides a detailed object diagram.

Finally, with Figure 19 a partly object model of the advanced search example is presented.

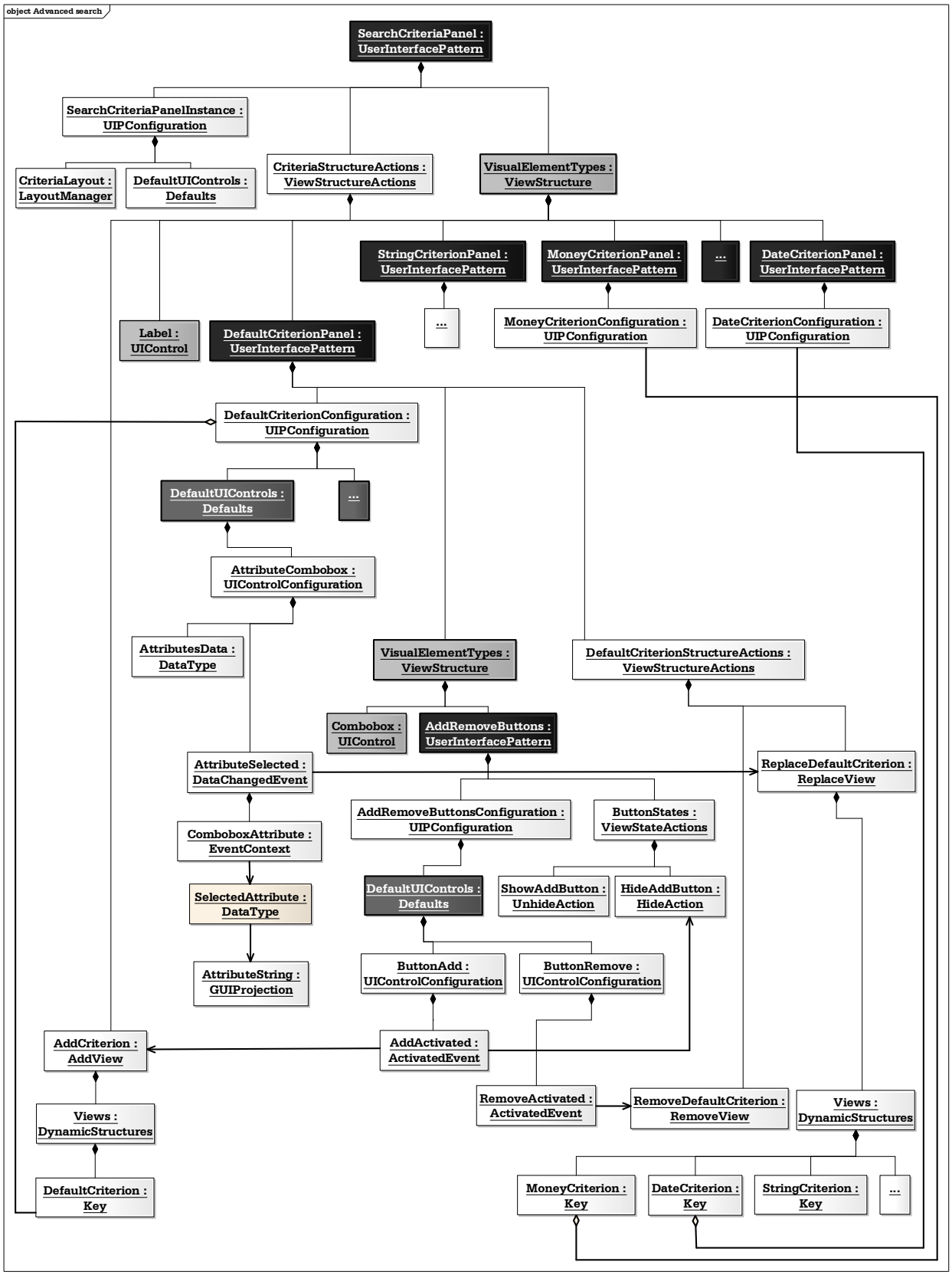

Figure 18. Advanced search example *SearchCriteriaPanel* object model.

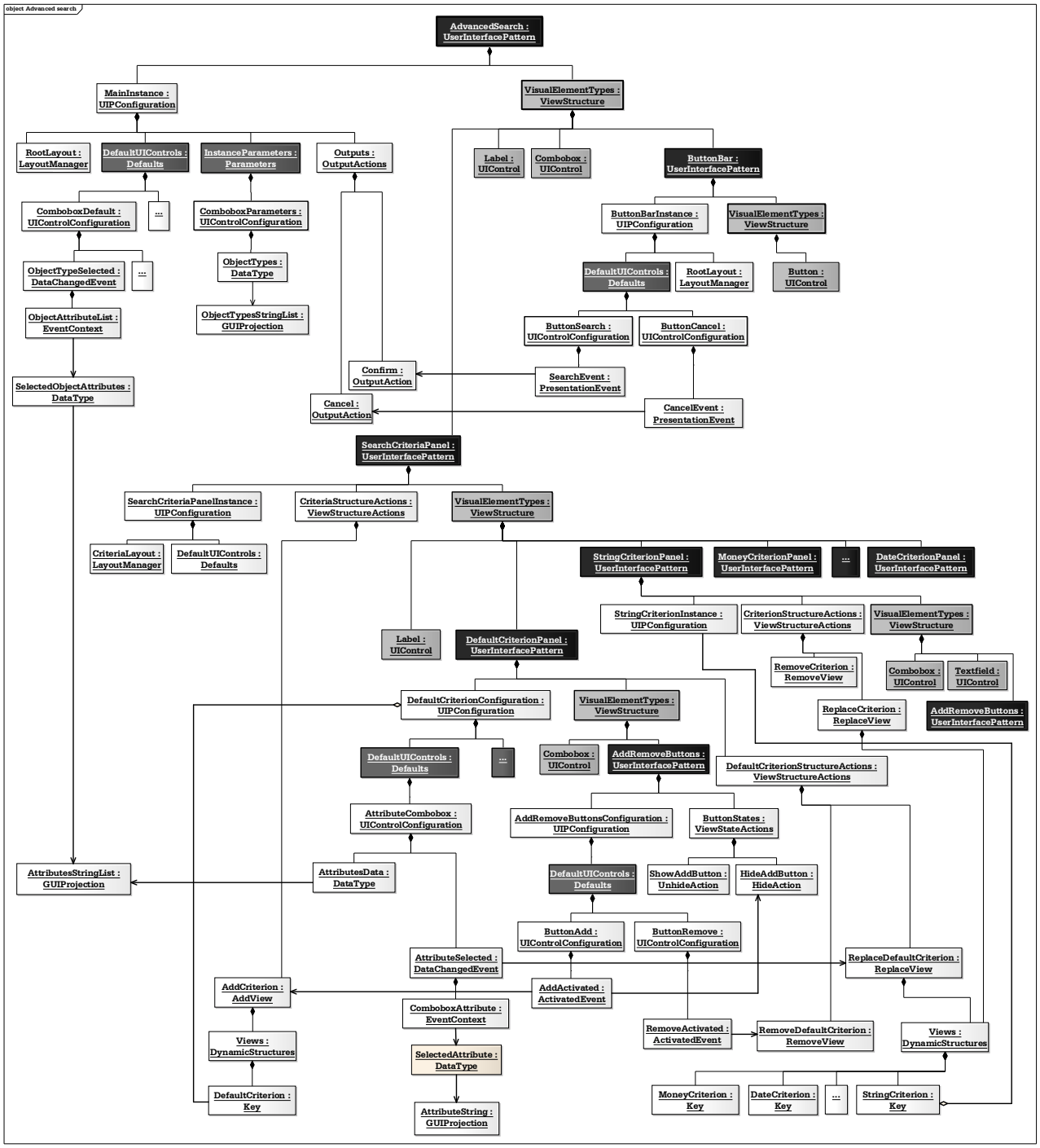

Figure 19. Advanced Search partly object model.

VIII. Results and Discussion

Achievements. With the elaboration of our analysis model, we detailed most factor impacts of our previous work on requirements for generative UIP representations [5][16]. Accordingly, we proposed fine-grained structures, which are in closer proximity to real applicable pattern notations than pure requirements can be.

Judgment. The current state of the analysis model is quite imperfect. However, with this initial iteration we achieved a better understanding of the information needed to express UIPs and their instances. A more vivid impression on requirements, which we have modeled explicitly and are implicitly supported by current approaches employing UIPs for model-based development [5], has been gathered. Furthermore, the model already may be used to verify the capabilities of notations for generative UIPs.

The object models gave us a good impression on the current state of the analysis model. Furthermore, the probing of the expression of selected example UIPs has proven a quite a good coverage of needed description elements. Much of the required information already could be modeled. However, the advanced search example is quite complex and could not be described here in more detail.

Moreover, the further analysis of the object models will reveal what impacts are yet to be enhanced. Not before the analysis model has been improved and object models can satisfactorily be expressed, we can think about a possible formalization of UIPs. This approach avoids unnecessary iterations and trial & error during the design of a dedicated UIP language.

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

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

Traceabilty. Concerning the realized impacts, we established traceability-links between the analysis model classes and the factor impacts of our requirements model. Figure 20 displays the relationship matrix accordingly. Please note that only generalized classes are included in the matrix. This is due to the fact that specialized classes do inherit the links from their parent classes.

As result, the analysis model almost fully complies with the elaborated requirements. Currently, "Hierarchical control flow for UIP compositions" and "Intercommunication events definition" do remain unsolved from the *control aspect* impacts. As far as the *view aspect* is concerned, the "Style definition" is an open issue.

Therefore, our analysis model has gained maturity and its elements may serve for the verification of modeling frameworks for generative UIPs on the presentation level. Thereby, the applied concepts, tool in- and outputs and especially the facilities of UIP formalization notation can be traced to the elements of the analysis model.

	Configuration of UIP context at design-time	Configuration of UIP context at run-time	Data-binding	Dialog action-binding	Encapsulation of UIP artifacts	Enumeration of elements	Hierarchical control flow for UIP compositions	Intercommunication events definition	Layout definition	Layout placement of elements	Naming of elements	Ordering of elements	Presentation action-binding	Style customization of elements	Style definition	Visual element structure definition	Visual element structure states definition
Caption											↑						
CommonParameters	↑																
DataType			↑														
Defaults	↑																
DynamicStructures		↑															
ElementConstraints																↑	
EventContext			↑														
Key		↑															
LayoutManager									↑								
LayoutPosition										↑							
Name											↑						
Order												↑					
OutputAction				↑													
Parameters	↑																
PresentationEvent													↑				
Style														↑			
UIControl																↑	
UIControlConfiguration	↑					↑											
UIPConfiguration	↑	↑			↑	↑											
UserInterfacePattern																↑	
ViewStateAction																	↑
ViewStructure					↑											↑	
ViewStructureAction		↑															
ViewStructureElement					↑											↑	

Figure 20. Traceability matrix: Factor impacts realized by analysis model classes.

Unsolved control impacts. Currently, our model only supports *ViewStructures*, which consist of UIPs always being in close cooperation. Nested UIPs are not yet intended to be reused outside the specification or their super-ordinate UIP. Being aware of this barrier, we may need to define facilities such as pattern interfaces, as this was proposed by both UsiPXML [8] and Seissler et al. [10]. In this regard, the *OutputAction* may be refined to accommodate the events required for UIP inter-communication. Eventually, the *UIPConfiguration* may be supplemented by certain input types. In the end, the first three *control aspect* impacts remain unsolved for now.

Open issues. We are aware that our model needs further elaboration and especially verification. Further issues to be solved persist in the classification and delimitation of UIP specification units. The relationships among UIPs discussed by Engel, Herdin and Märtin [23] may be considered, too.

IX. Conclusion and Future Work

Ultimately, we drafted an analysis model for UIPs by resuming our previous work on requirements towards a definition for generative UIPs. As result, the analysis model already covers mandatory structures and expresses variability aspects of generative UIPs. Together with our factor model, the analysis model may be taken into consideration for the verification of the capabilities of other UIP based approaches or languages mentioned and not mentioned here. Our object

model examples proved that current elements of the analysis model are required and sound for certain UIP instances.

However, the presented user interface pattern analysis model has to be reviewed and refined with the aid of other researchers in the future in order to establish a focused basis for a sophisticated generative pattern definition. With the latter, a dedicated notation can be developed that will allow the modeling of a vast and flexible range of UIPs. On that basis, a more thoroughly applicable solution concerning the covered GUI parts will be available in theory. In the long run, maybe the complete GUI system can be expressed with generative UIPs and their customized instances, even if this means to capture system specific or custom UIPs in the same specification format for the sake of a unified generative process. As time has revealed, several model-based GUI generation solutions and processes have emerged and came to their limits when the need for fully variable UIP instances came up. For generative UIPs, the path to a mature definition has been paved by our contributions.

In contrast, design patterns have already evolved over time and found a common notation and expression. This was feasible because the shape and aspects for that kind of patterns concentrated on the general abstraction of object-oriented paradigm, which is easier to grasp for developers as its reach in system responsibilities is very general and universal, and most important, limited to the concepts of a reduced set of repeating classes or their object instances. In addition, design patterns are applicable for many domains, and with reference to Figure 1, vast parts of their respective architecture artifact levels. For comparison, UIP modeling concepts tend to be bound to certain artifact levels governed by specific modeling frameworks. Thus, their UIP definitions are not flexible to allow for a combination with artifacts on varying levels. Simply put, the UIP pattern concepts have not reached a vast and general applicability as design patterns did.

Concerning the GUI architecture and patterns, there is no single shared definition or interpretation for MVC that can be relied upon. The pattern functions as a mental model layer on object-orientation for developers to classify the complex responsibilities and flows of GUIs by using atomic universal components customized for higher architecture understanding. Looking at our proposed analysis model, instantiable UIPs with their variable aspects in this abstract diagram are considerably more detailed and complex compared the initial MVC representations.

To conclude, we have to strive for a better understanding of UIPs, and after some iterations, a common UIP concept resembling the maturity of design pattern will be established finally. In the end, we have to enhance the available work on model based GUI development processes with the new UIP definition. Alternatives do not exist, as the presently available solutions offer no common generative UIP definition and thus can only cover a small portion of the GUI system, do not allow for sufficient variability or architecture artifact coverage. Again, we do need to strive for a common generative UIP definition in order to derive a conceptual basis the technical implementations and tools can be based on.

Future work. For future work, we see a refining and correcting iteration for the analysis model with regard to simplicity and completeness according to all impacts. In detail, we have to assess the mandatory and optional parameters on the basis of our listed examples. Furthermore, we will concentrate on the unsolved control aspect issues. With the progression towards an improved version of our analysis model, a more general applicable model-based UIP development process may be established in the future.

After completion of the UIP analysis and definition, we plan to consider options how to establish a notation that will be a realization of our analysis model. As candidates for GUI specification languages, UIML and UsiXML are likely to be considered for basic foundations. Eventually, for both languages enhancements will have to be developed in order to enable the support of generative UIPs.

REFERENCES

[1] S. Wendler and D. Streitferdt, “An analysis model for generative user interface patterns,” The Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 13) IARIA, May 27 - June 1 2013, Xpert Publishing Services, pp. 73-82, ISSN: 2308-3557.

[2] X. Zhao, Y. Zou, J. Hawkins, and B. Madapusi, “A business-process-driven approach for generating e-commerce user interfaces,” The Tenth International Conference on Model Driven Engineering Languages and Systems (MoDELS 07), 2007, Springer LNCS 4735, pp. 256-270.

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

[4] G. Meixner, F. Paterno, J. Vanderdonckt, “Past, present, and future of model-based user interface development,” i-com, vol. 10, issue 3, November 2011, pp. 2-11.

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

[6] R. Beale and B. Bordbar, “Pattern tool support to guide interface design,” Human-Computer Interaction - INTERACT 2011 - 13th IFIP TC 13 International Conference, Part II, Sept. 5-9 2011, Springer LNCS 6947, pp. 359-375.

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

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

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

[10] M. Seissler, K. Breiner, and G. Meixner, “Towards pattern-driven engineering of run-time adaptive user interfaces for smart production environments,” The Fourteenth International Conference on Human-Computer Interaction (HCII 11), Part I, July 2011, Springer LNCS 6761, pp. 299-308.

[11] K. Breiner, G. Meixner, D. Rombach, M. Seissler, and D. Zühlke, “Efficient generation of ambient intelligent user

interfaces," The Fifteenth International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 11), Sept. 2011, Springer LNCS 6884, pp. 136-145.

[12] M. J. Mahemoff and L. J. Johnston, "Pattern languages for usability: an investigation of alternative approaches," The Third Asian Pacific Computer and Human Interaction Conference (APCHI 98), 1998, IEEE Computer Society, pp. 25-31.

[13] A. Dearden and J. Finlay, "Pattern languages in HCI; A critical review," Human-Computer Interaction, vol. 21, issue 1, special issue: Foundations of design in HCI, 2006, pp. 49-102, http://www.tandfonline.com/doi/abs/10.1207/.U4hygHe8C4I.

[14] J. Borchers, "A pattern approach to interaction design," Conference on Designing Interactive Systems (DIS 00), August 17-19 2000, ACM Press, pp. 369-378.

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

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

[17] K. Breiner, M. Seissler, G. Meixner, P. Forbrig, A. Seffah, and K. Klöckner, "PEICS: Towards HCI patterns into engineering of interactive systems," The First International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 1-3.

[18] M. van Welie, G. C. van der Veer, and A. Eliëns, "Patterns as tools for user interface design," in Tools for working with guidelines, C. Farenc and J. Vanderdonckt, Eds. London: Springer, pp. 313-324, 2000.

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

[20] E. Hennipman, E. Oppelaar, and G. Veer, "Pattern languages as tool for discount usability engineering," The Fifteenth International Workshop Interactive Systems. Design, Specification, and Verification (DSV-IS 08), 16-18 July 2008, Springer LNCS 5136, pp. 108-120.

[21] S. Fincher, PLML: Pattern language markup language. http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html, 2014.05.30

[22] S. Fincher, J. Finlay, S. Greene, L. Jones, P. Matchen, J. Thomas, and P. J. Molina, "Perspectives on HCI patterns: concepts and tools (introducing PLML)," Extended Abstracts of the 2003 Conference on Human Factors in Computing Systems (CHI 2003), ACM, 2003, pp. 1044-1045.

[23] J. Engel, C. Herdin, and C. Märtin, "Exploiting HCI pattern collections for user interface generation," The Fourth International Conferences on Pervasive Patterns and Applications (PATTERNS 12) IARIA, July 2012, Xpert Publishing Services, pp. 36-44, ISBN: 978-1-61208-221-9.

[24] J. Vanderdonckt and F. M. Simarro, "Generative pattern-based design of user interfaces," The First International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 12-19.

[25] 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, issue 11-16, Proceedings of WWW8, 17 May 1999, pp. 1695-1708.

[26] 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, July 11-13 2004, Revised Selected Papers, R. Bastide, P. A. Palanque, and J. Roth, Eds. Heidelberg: Springer LNCS 3425, pp. 200-220, 2005.

[27] J. Vanderdonckt, "A MDA-compliant environment for developing user interfaces of information systems," The Seventeenth International Conference on Advanced Information Systems Engineering (CAiSE 2005), June 13-17 2005, Springer LNCS 3520, pp. 16-31.

[28] M. van Welie, A pattern library for interaction design. http://www.welie.com, 2014.05.30.

[29] Open UI Pattern Library. http://patternry.com/patterns/. 2014.05.30.

[30] A. Toxboe, User Interface Design Pattern Library. http://www.ui-patterns.com, 2014.05.30.

[31] 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.

[32] H. Umbach and P. Metz, "Use Cases vs. Geschäftsprozesse. Das Requirements Engineering als Gewinner klarer Abgrenzung," Informatik Spektrum, vol. 29, issue 6, pp. 424-432, December 2006, Springer, ISSN: 0170-6012.

[33] A. Wolff and P. Forbrig, "Deriving user interfaces from task models," Workshop Model Driven Development of Advanced User Interfaces (MDDAUI 09), Feb. 2009, CEUR Workshop Proc. Vol-439.

[34] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA: Addison-Wesley, 2004.

[35] F. Radeke, P. Forbrig, A. Seffah, and D. Sinnig, "PIM tool: Support for pattern-driven and model-based UI development," The Fifth International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 06), Oct. 2006, Springer LNCS 4385, pp. 82-96.

[36] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml. 2014.05.30.