A Software Category Model for Graphical User Interface Architectures

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

Abstract — **The development and maintenance of graphical user interfaces (GUI) for business information systems is still affected by software architectures lacking quality. Only basic patterns and few reference architectures are available for GUI development. There exist no standard architectures for reuse. High efforts accumulate for the adaptation of patterns but the resulting architecture quality often does not represent the desired separation of concerns and is hard to maintain. In this work, general GUI architecture design issues are analyzed. The foundation of the analysis is elaborated as a software category tree that represents the common responsibilities within GUI architectures. As result, the major design issues of GUI systems are summarized. To assess other GUI reference architectures, the software category tree may be of value.**

Keywords — GUI software architecture; software architecture; user interface patterns; graphical user interface.

I. INTRODUCTION

A. Motivation

Domain. Business information systems represent a domain that is largely influenced by software architecture considerations. Especially the graphical user interface (GUI) sub-system is likely to induce high efforts [1] for both development and later maintenance. This applies for both standard and individual software systems as a high demand for individually designed GUI systems is actually present.

Problem. However, GUI architectures are not standardized to the required detail, since historically applied patterns have not converged towards a detailed standard architecture governing every responsibility for change. In addition, the higher degree of system integration into business processes demands for explicit implementations of comprehensive requirement artifact types like use cases, tasks and business processes. Those have to be integrated with rather old patterns like MVC [2] and its variants [3], which did not consider such deep and vast requirements basis. Reference architectures [1][4] and several patterns (design and architectural) [5][6] have been suggested, but have not been properly integrated with traceability [7][8] concepts to keep track of requirements. Moreover, GUI frameworks have a large impact on the structure and often cannot be isolated properly to separate technical implementations from domain or project specific requirements.

Consequences. When systems have grown after several maintenance steps, different concerns tend to be mixed up within the GUI architecture the larger the requirements basis is and the more complicated the integrated frameworks are. For instance, application server calls, data handling, task and dialog control flow can no longer clearly separated in the software architecture. Finally, the GUI and application sub-systems cannot be separated easily and the evolution of both depends on each other. Business logic tends to be scattered in the GUI dialogs [9] and the "smart UI antipattern" [10] may become a regular problem. The architecture was layered during design phase, but the encapsulation of components and separation of concerns did not prove in practice [9]. This is maybe due to used frameworks that expect a certain architecture, which alters original design. More likely is the phenomenon that the architecture was based on common patterns and reference architectures that could not be refined in time with respect to desired quality and extensibility. Lastly, the two concluding points from Siedersleben [9] are still of relevance: standardized interfaces between layers are still missing and technical frameworks still dominate the architecture and evolution. Currently, there are even more than three layers in business information systems and the segregation got even more complex.

User interface patterns. Current research is occupied with the integration of a new artifact type in the development of GUI systems. Being based on pattern concepts, user interface patterns (UIPs) have been approached [11][12][13] to facilitate the generative development of GUIs and highly increase the reuse of proven visual and interaction design solutions that originate from descriptive human computer interaction patterns [14][15]. According to the generative nature of these attempts, the development of GUIs shall be shortened by model-based sources that specify both the GUI system's view instances and the coupling between functional related and GUI-system-architecture components.

Current limitations. Currently, there are still design issues within GUI patterns or reference architectures that hinder the evolution and maintenance of existing systems. To establish a target software architecture of high quality for the implementation of UIPs, these issues have to be addressed in the first place. In fact, UIPs need a clear basis of reuse: an architecture with well separated concerns that permits the flexible allocation and exchange of these greater units of design. Whether UIPs will be generated, interpreted or provided by a virtual user interface [16][17] the resulting architecture will be at least as complex as for standard GUIs. So, the common issues in design will prevail and affect UIP based solutions.

B. Objectives

To prepare the integration of UIPs into GUI architecture and at the same time preserve their reusability and variability in different contexts, open issues in GUI architecture development have to be identified and solved. Therefore, our

 ISBN: 978-1-61208-367-4

goal is to provide a detailed analysis of these open design problems. Hence, we will have to identify the re-occurring responsibilities of GUI architectures and their relationships. On that basis, the frequent applied MVC pattern is reviewed. In addition, we will analyze the Quasar client reference architecture [1] that provides more detail than regular patterns and was created especially for the domain.

C. Structure of the Paper

The following section provides descriptions of common patterns and reference architecture considerations for GUIs. In the third section, we will elaborate a general responsibilities model for GUI architectures. In Section IV, the GUI architecture patterns are reviewed. The results are summarized in Section V, before we conclude in Section VI.

II. Related Work

A. Architecture Patterns for Graphical User Interfaces

With the invention of object oriented programming languages, a clear assignment of the cross-cutting concerns, which are common for a GUI dialog, had to be enforced. Eventually, the model view controller pattern was introduced [2] that distinguishes three object types as abstractions to accept defined responsibilities.

In Figure 1, we present a possible architecture application diagram of the MVC pattern. Generally, the MVC pattern promised a separation of concerns, flexibility and even reuse of selected abstractions. From a practical point of view, the classic MVC pattern misses many details that are essential to fulfill these claims. In this regard, the pattern leaves the task to decouple the three abstractions to be solved by the developer. It is noteworthy that the *Controller* is in charge of many responsibilities at once. Both the handling of technical events (*PresentationEvent*) and the initiation of the final processing of data by the application kernel (*ApplicationKernelService)* are governed by the *Controller*. Therefore, this design unit is closely coupled to the *View*, as well as to the *Model*. As far as the *View* is concerned, the structure of the *Model* has to be known to enable the update of defined UI-Controls via *DataRead*.

There exist many sources of the MVC pattern [18][19]. A widely accepted description can be found in [6].

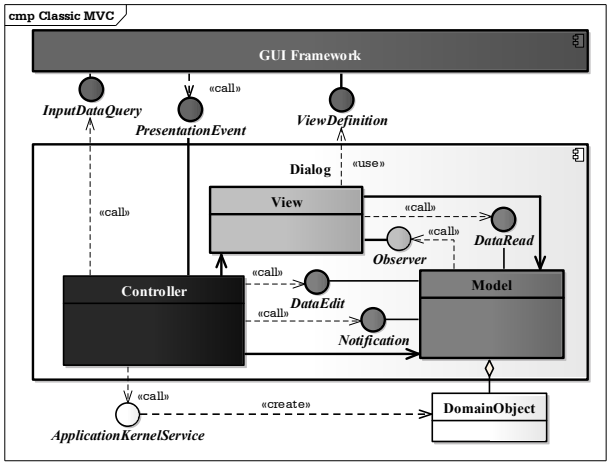

Figure 1. A common MVC architecture pattern variant.

To cope with the close coupling and missing details, several variations of the MVC have been discussed [3][20]. In general, the variations in design differ concerning the distribution of responsibilities among the three abstractions. Several more patterns [5][6][19] occurred that mainly altered the control or introduced new concerns and abstractions. Nevertheless, they fulfill the same purpose of guiding the identification and modularization of classes in object-oriented GUI architectures.

B. Graphical User Interface Event Processing Chain

To be able to discuss the GUI responsibilities with increasing detail, we would like to refer to the conceptual model of event processing within GUI architectures as described by Siedersleben [21]. In Figure 2, a variation of this model is displayed. Thereby, technical events will be emitted from the operation system or later the *GUI Framework* when the user has interacted with a certain GUI element. Within the architecture, the event is either processed or forwarded by the individual components depicted in Figure 2.

It is notable that there is a distinction of events inside the *Dialog* component. For reasons of separation of concerns, and ultimately, better maintenance of systems, the *Presentation* was assigned responsibilities with a closer connection to the technical aspects of the *GUI Framework*. Accordingly, the *Presentation* is in charge of governing the layout of the current *View* and applies changes in layout, e.g., mark the UI-Controls where entered data failed the validation or activate panels when current data state requires for additional inputs. In contrast, the *DialogKernel* is to be kept independent from any technical issues as far as this is possible. So, the latter is assigned to communicate with the *ApplicationKernel* and its components instead.

By flowing all the way from the *Operating System* towards the *Application Component*, a tiny technical event may result in the initiation of greater operations inside the *DialogKernel* or even *ApplicationComponent*. That is why Siedersleben speaks of a "value creation chain" [4][21].

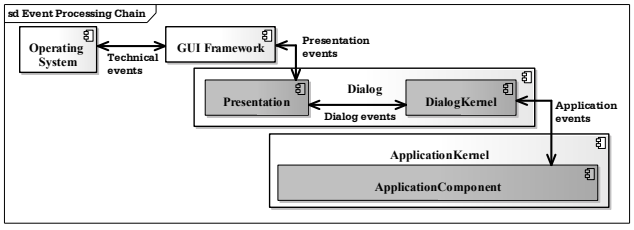

Figure 2. Value creation chain of graphical user interfaces derived from [21].

C. Standard Architecture for Business Information Systems

Siedersleben and Denert tended to the issues of close coupling and a better separation of concerns for GUI architectures in [16]. The main goal of their attempts was to improve the general quality of the software architecture of business information systems. With respect to the GUI, they made suggestions [16] that would prepare the standardization of the architecture of the particular domain.

Quasar. Siedersleben pushed towards further standardization attempts concerning the GUI architecture of business information systems. His efforts culminated in the creation of the quality software architecture (Quasar) [4]. Acclaimed design principles and architectural patterns, as well as the vast usage of interfaces for decoupling in combination with a new instrument for component identification were incorporated into a single software architecture manifest, which was intended to become the domain's standard.

Parts of a reference architecture [1] and the object-relational mapper Quasar Persistence have been published. Conversely, the main ideas of standardization were neglected in [1] and reference architecture elements should fill the gap.

Software categories. As far as the component identification is concerned, so called software categories were introduced. They consist of the five categories *0*, *A*, *T*, *R* and *AT*. *0* designates elements that are reusable in any domain like this is applicable for very basic data types a programming language would offer. *A* software is dedicated to implement a certain domain's requirements, meaning particular functions like the calculation of target costing or the scheduling of production plans for a certain machinery. In contrast, *T* software is responsible for the integration of technical aspects like data bases and GUI frameworks. *R* software is needed whenever a technical data representation has to be converted for processing with *A* software types, e.g., a GUI string type describing a book attribute is converted to a ISSN or ISBN. In fact, *R* software also is *AT* software per definition as both domain specific and technical knowledge or types are mixed up. Thus, *AT* software should be avoided and would be an indicator for the quality of the implementation or architecture. Only the *R* software used for type conversions would be permitted.

GUI reference architecture. Concerning the reference architecture portions of Quasar, the GUI client architecture [1][4] has to be mentioned for the scope of our work. The main parts of that architecture are illustrated by Figure 3 that is derived from [4], since this is the most detailed source available. The interface names in brackets resemble the original but not very descriptive designations. The unique elements of the Quasar client architecture are the following three aspects.

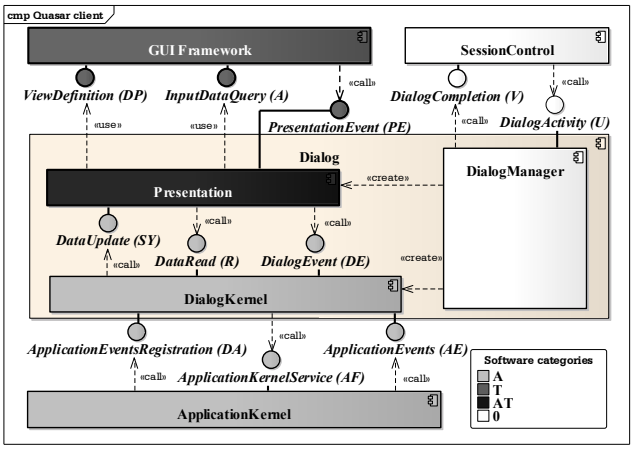

Figure 3. The Quasar client architecture based on [4].

Firstly, there was made a distinction of presentation and application related handling of events; the basic concept of the "value creation chain" introduced in Section II.B was developed further. Thus, there are the two design units *Presentation* and *DialogKernel* that resume original MVC *Controller* tasks besides other ones. The software categories mark both units according to their general responsibilities. The *Presentation* possesses the knowledge how certain data is to be displayed and how the user may trigger events. In contrast, the *DialogKernel* determines what data needs to be displayed and how the application logic should react to the triggered events. The communication between them is exclusively conducted via three *A* type interfaces.

Secondly, the Quasar client introduces relatively detailed interfaces and communication facilities between components compared to other GUI patterns. To be able to fulfill its objectives, the *Presentation* relies on the *ViewDefinition* interface to construct the visual part of the dialog. Via *InputDataQuery*, the current data stored in the technical data model of respective UI-Control instances can be altered or read by the *Presentation*. Events emitted from UI-Control instances are forwarded to the *Presentation* with the operations of *PresentationEvent*.

The interfaces between *Presentation* and *DialogKernel* are mainly concerned with event forwarding and the synchronization of data between both components. In detail, *DialogEvent* is called by the *Presentation* whenever the *DialogKernel* has to be notified of an event relevant for application logic processing, e.g., a command button like OK or a search for available data was initiated. The Quasar client foresees two options for data synchronization. This communication step is essential, since both components possess different knowledge, and thus, work with different data structures, what is marked by the different software categories. Either the *Presentation* could read current data via *DataRead* or the *DialogKernel* would update the *Presentation* by the means of *DataUpdate*. This design shall decouple the application logic from technical aspects found inside *Presentation* and its interfaces for interaction with the current *GUI Framework*.

Thirdly, aspects that are concerned with surrounding components are also described with the Quasar client. These are interfaces dealing with the construction, deletion of dialog instances (*DialogActivity*) and reporting of results (*DialogCompletion*). Furthermore, a DialogKernel can register for notification (*ApplicationEventsRegistration*) about events (*ApplicationEvents*) originated from *ApplicationKernel*. For creation of value relevant for business logic, the interface *ApplicationKernelService* is called by the *DialogKernel*. There are more interfaces available for the coordination of transactions and the checking of permissions via an authorization component. For more details, interface specifications and a dynamic view on the architecture, please consult [1].

III. GENERAL GUI RESPONSIBILITIES MODEL

A. Approach

As the basic GUI patterns and the Quasar client reference architecture are too abstract and general to describe detailed responsibilities required for implementation purposes, we

will establish a fine-grained responsibilities model based on the software category instrument suggested by Quasar. The software categories are intended to refine tasks and fill gaps left open by the available patterns. Thereby, the categories will represent an ideal model with least coupling that allows for planning dependencies among potential units of design.

Consequently, we need to establish a basis for the responsibilities that are regularly discovered in a GUI architecture. Eventually, we follow the approach to investigate on relevant responsibilities mainly from related work, other known sources [2][3][4][6][16][18][19][20][21][22] and own experiences. In fact, we do a decomposition of GUI architectures to rather atomic functions. These functions will be separated and delimited in order to establish a unique software category tree. We examine, what can be solved with *0* or *A* software and what concerns are definitely dependent on GUI framework code.

When common GUI architecture responsibilities have been identified and systematically analyzed concerning their dependencies, the potential interfaces for communication between components or classes can be derived. According to Quasar [4], an interface ideally should be defined on the basis of a software category that serves as a parent for both categories to be linked. Thus, the identification of design units and their interface structure requires some planning.

B. Quasar Software Categories Reviewed

The concept of the Quasar software categories is ambiguous. They promise to be an instrument for component identification and quick software quality assessments. Nevertheless, they were not provided along with a clearly defined method for their proper definition or application.

The software category types defined by Quasar can be applied for the very basic valuation of architectures, since they symbolize a very rudimentary separation of concerns between neutral, domain and technical related concepts. The further and project relevant refinement of the basic categories will eventually lead to a much more powerful representation of design criteria like cohesion and coupling or design principles like modularization as well as hierarchy. In this regard, “concerns” represent heavily abstracted requirements and related functions. Siedersleben [4] states that each software category ideally acts as a representative for a certain delimited topic. Consequently, the preparation of components with the aid of software category trees shall help to create high cohesive and encapsulated design units.

Traceability. On that basis, software categories will be used to judge the purity of traceability-link [7][8] targets, meaning that the artifacts will be examined with respect to their responsibilities. When a target is made up of a mixed category, in the worst case *AT*, then it will be considered either as a model lacking detail or a design that is harder to maintain, since the developers will eventually separate the concerns during implementation by themselves. The latter is a major aspect besides the identification of potential components; that is why we consider software categories as a relevant marker. In sum, software categories can be useful to reduce the complexity while tracing requirements to design: the categories could be kept in order to mark certain design elements inside traceability-metamodels, which are outlined in [8]. Thus, the general or refined responsibilities of design elements will be visible, so traceability-link targets can be more detailed.

A major problem lies in the definition and segregation of software categories. It was not clearly defined what elements drive the creation and delimitation of a software category. According to known sources [4][9], this might either be specialized knowledge how to handle certain algorithms and data structures or dependencies of an entity.

C. Rationale on Software Category Practical Application

Basic software categories. As the software categories are not clearly defined in original sources, we will have to point out how to create new and delimit existing software categories. On the root level, we will comply with Quasar and use the basic categories *0* (white), *A* (light grey), *T* (medium grey with white caption) and *AT* (dark grey with white caption). The basic category *Construction and Configuration* was added to represent the creation of new objects as well as the configuration of interfaces with implementing objects. On the next level, layers and technological boundaries of the application architecture are represented. *Presentation* and *Dialog Logic* were separated as categories according to the event processing of Figure 2. Our aim was to provide a software category tree with separated concerns to describe a complete decomposition of GUI architecture aspects.

As the tree gets more detailed, categories will become very fine grained and embody components, classes or even operations. Since the categories can distinguish components and their dependencies, they could be applicable for the delimitation of the smaller units of design, too.

Category identification. To identify each of the following categories, we applied several rules of thumb. During the analysis of GUI architectures, we derived categories from the different families of operations that regularly occur. In general, these were the definition or modification of new entities or their properties, event triggering or processing, as well as forwarding of both data and events. These kinds of operations occur for different contexts like technical or application related objects of general GUI pattern components that are common for MVC or the Quasar client. The different contexts symbolize certain levels in the software category tree and were derived from reasonable abstractions like application logic, abstract presentation and presentation technology. We distinguished the belonging operations and data structures according to the knowledge and types required for their processing. When operations demanded for the usage of certain types in a context that was not in scope of the originator, then categories were definitely of a mixed kind. In contrast, categories were left pure when interfaces using neutral *0* types could be used for delegations. A hint close to implementation considers what would be the import declarations in a unit of design with respect to Java language. If the import was based on interface types using neutral *0* types, the category would remain pure. The category would be mixed, if the imports will demand for the addition of types defined exclusively in the imported unit of design.

 ISBN: 978-1-61208-367-4

D. Graphical User Interface Software Category Model

The resulting software category tree is depicted in Figure 4 and will be developed in the following paragraphs. It has to be considered that the categories do model dependencies between units of design and no flow of events or algorithms. Although there will be interfaces between categories for later implementation, these cannot be illustrated by the category tree but will be determined concerning the possible type. According to Quasar [4], two different categories may communicate via types that originate from a shared parent category.

The main categories *Application Kernel*, *Dialog Logic*, *Presentation* are *A* category children, since they depend on the individual requirements of a software system.

Presentation. The categories derived from *Presentation* are closely related to the view and controller of the MVC pattern [6] and detail both their responsibilities.

Presentation is marked with FUI (final user interface) [23] given that this category symbolizes the certain knowledge required for creating the specific view part of a given GUI system. This category is further branched into *View Definition* and *Presentation Event Handling*. The involved categories have to comply with project specific dialog specifications and at the same time need to possess knowledge about the types and operations the involved *GUI Framework* offers. Hence, all sub-categories heavily depend on technical aspects. They each constitute a mixed category.

The *View Definition* category is detailed with the responsibilities required for the initial creation of the visual parts of a dialog and the declaration of layout specific elements. We separated the *Layout Definition* and *UI-Control Configuration* as the layout aspects often involve the usage of dedicated objects and operations that considerable differ from the instantiation and configuration of UI-Controls. For the reasons that events require dedicated operations and not all created UI-Controls have to be bound to certain events, the category *Action Binding* was separated as a specialization of the *UI-Control Configuration*.

The *Presentation Event Handling* category serves the task to deal with *Presentation events* according to Figure 2 and is branched into *Presentation Data Handling*, *View State Changes* and *Event Forwarding*. The first child handles both the reading (*Model Data Observer*) and editing (*Model Data Edit*) of dialog data from the *Presentation* perspective. The changes in layout, properties and arrangement of active UI-Control instances during runtime are optional tasks that are embodied by the category *View State Changes* and its children. Certain events cannot be further processed by the visual dialog units, so that they need to notify the next unit in the chain of responsibility. This rationale is based on Figure 2. The required knowledge about the respective events and forwarding commands is encapsulated by *Event Forwarding*.

GUI Framework. As far as the *GUI Framework* is concerned, we decided for the distinction of layout and UI-Control specific knowledge or types. The *UI-Control Library* implements all operations and types that are required for the instantiation of any available UI-Control, the modification of its properties (*UI-Control Properties*) and the definition of its data content (*Technical Data Models*). Often there are various data types with different complexity associated to the available UI-Controls of a framework. They need to be handled by the *Presentation Data Handling* category in order to store and retrieve data in the specific formats like lists, trees, text areas or table grids.

Dialog Logic. The last main category that is to be placed in the vicinity of a dialog is the *Dialog Logic*. Categories that are involved in the data structure definition and its logical processing refine the *Dialog Logic*. The basis of these categories is provided by the Quasar client [1][4] and the model part of the MVC pattern [6]. In analogy to the *Presentation* category, we distinguish the definition of data objects (*Dialog Data Model*) with associated operations and the event handling (*Dialog Event Handling*).

The category *Dialog Data Model* depends on knowledge about the *Domain Data Model* defined by the *Application Kernel* as well as *Data Queries* that may deliver the composition of selected attributes from different entities in order to create new aggregates relevant for display. The *Data Queries* category belongs to the *Application Server Calls* category, which encapsulates knowledge about the available application services, their pre-conditions, invariants and possible results with respect to the dialog logic.

The *Dialog Logic* category graph mostly constitutes pure *A* category refinements. However, the *Data Conversion* category is of mixed character. To define data structures that can be used in close cooperation with the *Application Services*, it needs to know about *Dialog Data Model,* and thus, incorporates its dependencies to the *Data Queries* and *Domain Data Model*. Besides, the *Data Conversion* category has to be aware of the current *Technical Data Models* in order to provide access for *Presentation Data Handling*. The latter has to know about the structure of defined data models (*Dialog Data Model* and *Technical Data Models*) to be able to delegate proper updates in both directions.

Event processing. The entire event processing chain and its association to software categories was challenging; our rationale will be explained as follows. Foremost, logical and presentation states were separated: Application logic tends to be stable (enter data, evaluate, present suggestions, make a choice and confirm), is traced to functional requirements, and thus, should be decoupled from GUI specifications. Although the flow of application logic is unaffected, the GUI and its technology supporting the user in his tasks may be altered several times starting with updated specifications and ending with the deployment of different *GUI Frameworks*. Additionally, the *Presentation* can be further differentiated into abstract visual states that have a close connection to the current application state and technological or concrete presentation states, which implement the former. The latter is translated to GUI UI-Controls via *GUI Framework* and its sub-categories. As result, we identified three major categories for state control to be considered below.

The *Dialog Event Handling* tree governs the application logic part of a dialog and has no concrete visual representations or related tasks. In contrast, it assumes the *Presentation* to maintain appropriate visual representations, but these remain abstract for the *Dialog Event Handling*, e.g., a view for data input is activated, data input was completed or current data leads to another view state for data input.

 ISBN: 978-1-61208-367-4

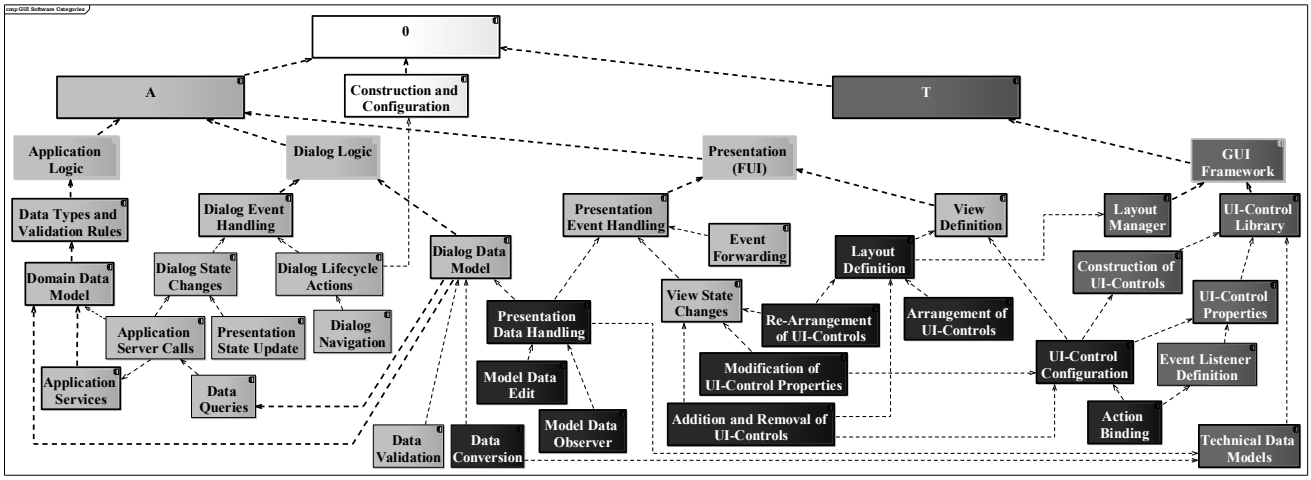

Figure 4. GUI responsibilities arranged as a software category tree.

From the application's perspective, a dialog may adopt different states during runtime. The required knowledge to control these states is represented by the category *Dialog State Changes*. Furthermore, this category is separated into categories, which either interact with the *ApplicationKernel* or the *Presentation*. Both its categories reflect the two general situations that may occur in any dialog: *Application Server Calls* may be initiated or a *Presentation State Update* can be triggered. The parent category *Dialog State Changes* possesses the knowledge how to react in a given situation. Its children are dedicated to solely trigger the required change of state that either addresses the *Application Server* or *Presentation*, which provide the state change execution.

Figure 5 provides an overview of possible interface connections between software categories involved in event processing. Please note that the interfaces need to be of the basic *A* category type as this is the common parent category of the displayed interacting categories.

The general flow of events is the following: initially, the user triggers some events that may be forwarded to *Dialog Event Handling* for further evaluation. Depending on the current state of the dialog, *Dialog Lifecycle Actions* (creation and deletion of dialogs and their objects), *Application Server Calls* (commit a sequence of service calls), a *Dialog Navigation* (change of current view or the instantiation of sub-dialogs) or a *Presentation State Update* (change of the visual representation) may be delegated. In this regard, the key design issue is that the *Presentation* has no knowledge in its sub-categories how to decide on a proper reaction for events relevant for dialog logic. Therefore, the event firstly is forwarded via the topmost interface in Figure 5. Then, the *Dialog Event Handling* evaluates the event and delegates to one of its children, which further delegates to the displayed interfaces in Figure 5 and initiates the final change of state. Concerning the *Presentation State Update* in Figure 5, either a *Dialog Navigation* (separate dialogs or an auxiliary search dialog are instantiated) or *View State Changes* (panels, wizard steps or tabs are switched) are committed via interfaces. In this context, the knowledge when to trigger any of the interface operations is kept in the children of *Dialog Event Handling* with a white border in Figure 5. In contrast, the execution of the respective state change is encapsulated in the categories that implement the interfaces. At last, the state changes are completely decoupled from the point in time when they are requested. Finally, the *Presentation Event Handling* is separated into event processing that is either concerned with data or the visual structure. Mostly the data relevant events can be processed locally by the *Presentation* if no forwarding is registered. However, the *View State Changes* do require the forwarding of events to the *Dialog Event Handling* first, before they can be committed. This is due to the decoupling of view states and their better exchangeability. Moreover, the differentiation of event evaluation, triggering and state change execution supports the reuse and change of views as they are better decoupled from dialog logic components. In this regard, view states are relevant for the *Dialog Logic* but not their concrete appearance, which can be adapted frequently.

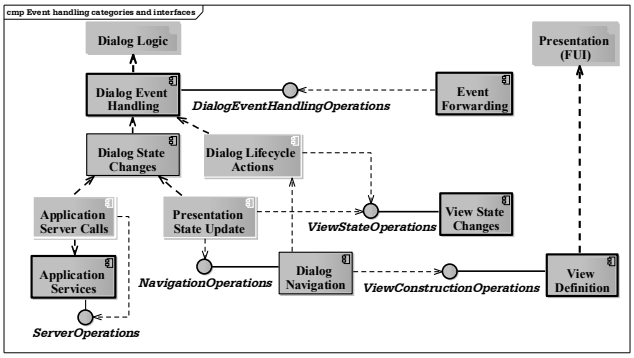

Figure 5. Software categories relevant for event processing and possible interfaces.

IV. Review of GUI Architecture Patterns

In this section, we review the presented GUI patterns of Section II in the light of the elaborated software categories.

A. MVC Variants

For the review of classic GUI architecture patterns, we would like to refer to exemplary and valuable work published in [3] and [20], which is valuable for filling gaps and giving directions for related design decisions. Therein,

options for refinement and customizing MVC based architectures are proposed and discussed. It is still up to the developer to decide on the several choices. In contrast, the Quasar client architecture presents a reference for our domain that already has some refinements incorporated.

Positive aspects. Both patterns and Quasar client share two positive aspects that motivate their application. Firstly, the data storing component does not depend on any other of the components, and so, can independently evolve. Secondly, only one of the components resumes the task to call *ApplicationKernel* services. This aspect eases the design efforts for interfaces and data exchange formats between dialogs and *ApplicationKernel*.

Issues. According to the MVC variants, we see two major issues that will be described as follows.

Separation of concerns. Firstly, the degree of encapsulation and separation of concerns of MVC variants is very limited. There is no variant that is able to reduce the dependencies of all three abstractions altogether. Solely, the distribution of tasks is altered, and so, the visibility among components changes accordingly. In the end, one component will be assigned responsibilities that originate from the two other components as they are defined by classic MVC. Therefore, the component with concentrated tasks tends to be overburdened, and finally, can end up as the bottleneck from a maintenance perspective. Additionally, in certain variants the altering the tasks of the three components may result in a simplification of one component that can only be employed for stereotype tasks. There seems to be no ideal separation of concerns among three components.

In general, there are no hints given how the business logic and its related display can be decoupled. More precisely, the *View* part is directly coupled to the *GUI Framework*. In addition, the knowledge of the *View* has to constitute of how to operate the *GUI Framework* facilities (to construct the visual dialog parts) and what layout as well as what selection, order and arrangement of UI-Controls are needed to embody the domain and the current service in use.

Event differentiation. With regard to the event processing chain of Section II.B, the patterns do not distinguish clearly between events related to technical or application related concerns. In general, a guideline is missing for the decision when to shift between presentation or application related processing of events. So, the developer has to refine the architecture by himself. The reuse may be affected, since the *Controllers* end up processing both types of events for the sake of quick release cycles.

Cohesion. Concerning the identification of possible instances and their proper size, there are hardly any hints when to create new dialog instances or MVC-triads. Thus, the modularization of dialog components is to be done on behalf of the developer. Only the HMVC [24] gives some rudimentary hints. The general size and scope of MVC units is not clear. According to Karagkasidis [20], a *View* may constitute of single UI-Controls (widgets), containers like panels with a certain set of UI-Controls or whole dialogs.

Coupling. With respect to the limited separation of concerns more issues arise. The control of *Presentation* states and the handling of application related events to initiate *ApplicationKernel* service calls are closely coupled to *View* elements. Usually, in many MVC variants *Controller* and *View* maintain a strong dependency where the *Controller* is fully aware of the UI-Controls of the *View*. In fact, both components build an aggregated unit of design that cannot be reused and is harder to maintain. Eventually, a *Controller* can only interact with *Views* that comply with a certain set of states. Whenever the set of UI-Controls changes the possible states of the dialog alter as well, so that the *Controller* implementation may have to be revised each time.

To partly resolve this issue and decouple the *Controller* from application aspects, a developer could revert to the "Model as a Services Façade" [3] MVC variant. The *Model* would be assigned both data structures and related service calls for interaction with the *ApplicationKernel*. This step would raise a comparative discussion as whether it is favorable to build a separate service layer [25] or use the domain model pattern [19] exclusively for the structuring of the *ApplicationKernel*. In our opinion, the *Model* should not act as a service façade, since it would make parts of an *ApplicationKernel* service layer obsolete. According to the resulting dependencies to functional requirements, the traceability-links of use cases or tasks would be scattered among different *Models* and parts of the *ApplicationKernel*. Furthermore, the operations of the Model would be closely coupled to a certain data structure so that a *Model* cannot be easily combined with other application services in the future. Lastly, services should prevail, since there might be other clients besides a particular GUI to rely on services. There are more disadvantages with that solution like the stereotype character of the *Controller* [3], which will only serve a certain pattern of interaction. Thus, the *Model* should only contain data-relevant operations (getter, setter, aggregation and conversion, a state of current selection state, validation) and be reusable with other services. In this regard, the *Model* should act as a mere preparation of a data structure that is useful in the context of *View*.

Summary. The MVC and its derivates require much adaptation in order to be prepared for implementation [22]. The above mentioned issues considerably may have a negative impact the resulting architecture quality. The available patterns are definitely not easy to interpret with respect to the much more responsibilities illustrated by the software category tree in Figure 4.

The tracing of functional requirements to the parts of the GUI which coordinates *ApplicationKernel* will largely depend on the refinements the developers have incorporated. The resulting architectures will be heterogeneous and may add complexity to quickly provide an adapted solution for the particular domain. As long as there are no standard architectures or standardized responsibilities available, the developer is left with many choices that potentially will lead to vast differences in software architecture quality. The improved segregation of software categories in component architectures is goal hard to achieve with available patterns.

B. Quasar Client Reference Architecture

1) General Valuation

The Quasar client architecture provides the most detailed architecture view on GUI systems published so far and can be regarded as a refinement of the common GUI patterns.

Positive aspects. In contrast to the MVC variants, the Quasar client separates *Presentation* and *DialogKernel* as principal dialog components. This separation is the main source for its virtues, since more clearly distinguished *Controller* tasks are achieved. In this regard, the *Presentation* is required to handle technical events and the *DialogKernel* will process application related events in close cooperation with the *ApplicationKernel* services.

States and control. According to Siedersleben [4], the *Presentation* and *DialogKernel* components share a common structure: both possess memory for storing data, states and a control. Thus, both components are able to manage their states independently. A change of layout aspects in the *Presentation* would not affect the *DialogKernel* accordingly.

In theory, the changes of states are implemented in each component individually and can be triggered by *A* typed interfaces that may be designed on the basis of a command [5] pattern [22]. Consequently, the *DialogKernel* does not require knowledge about the inner structure of the *Presentation* and vice versa. Thereby, the *Presentation* may provide a set of operations that alter the layout of a dialog depending on the current content of data collected via *DataUpdate* interface. The triggering of visual state changes on behalf of the *DialogKernel* (*Presentation State Update*) may be possible but is not considered. For instance, a *DialogKernel* was notified via *DialogEvent* that the user has selected an item in a table listing available products. But the product is on back-order, so the *Presentation* should receive the command to display a certain state of the button bar, e.g., deactivate the "add to cart" button. Besides, a *DialogKernel* could be able to coordinate the inputs of a user working with two *Presentations* simultaneously.

2) Traceability-Links to GUI Software Categories

To be able to better valuate the Quasar client architecture, we traced the identified software categories of Section III.D to its structural elements. Figure 6 displays the resulting traceability matrix. The sources for traceability-links constitute software categories of varying detail arranged on the left hand side. Please note that the general parent software categories were excluded, since all child categories are presented in the matrix. On top of the matrix, the traceability-link targets are represented either by the components or interfaces of the Quasar client. Components not relevant as traceability-link targets were excluded.

Interpretation. We need to provide directions about the treatment of interfaces and connected dependencies, which are depicted in Figure 3. A client that imports and calls a foreign interface must have knowledge about the proper usage and sequences of operations. In fact, the deeper and more chained the commands are the more likely is the mixture of categories. Finally, the client will be dependent on the same software category the interface is composed of. This particularly applies to the *Presentation* (obviously an *AT* component) that extensively uses the *GUI Framework* interfaces, which are to be included in the traceability matrix. In contrast, single commands of abstract or stereotype nature like notify calls can be realized with a *0* type interface. Yet, the interfaces pose hard to valuate concepts as they inspire a dynamic view on the architecture like the sequences of commands or flow of algorithms. Ultimately, the interface operations would need further refinement for a final valuation. Partly, the Quasar reference architecture provides basic sequences for interfaces in [1].

	ApplicationEvents (AE)	ApplicationEventsRegistration (DA)	ApplicationKernel	ApplicationKernelService (AF)	DataRead (R)	DataUpdate (SY)	DialogActivity (U)	DialogCompletion (V)	DialogEvent (DE)	DialogKernel	DialogManager	InputDataQuery (A)	Presentation	PresentationEvent (PE)	SessionControl	ViewDefinition (DP)
Action Binding													↑	↑		↑
Addition and Removal of UI-Controls													↑			↑
Application Server Calls			↑	↑						↑						
Application Services			↑	↑												
Arrangement of UI-Controls													↑			↑
Data Conversion													↑			
Data Queries			↑	↑						↑						
Data Types and Validation Rules			↑													
Data Validation			↑							↑						
Dialog Data Model										↑			↑			
Dialog Lifecycle Actions							↑	↑			↑					
Dialog Navigation							↑	↑			↑				↑	
Dialog State Changes	↑	↑								↑						
Domain Data Model			↑													
Event Forwarding									↑							
Layout Definition																↑
Model Data Edit													↑			
Model Data Observer					↑	↑				↑			↑			
Modification of UI-Control Properties																↑
Presentation State Update													↑			
Re-Arrangement of UI-Controls																↑
UI-Control Configuration																↑

Figure 6. The GUI software categories traced to Quasar client components and interfaces.

Separation of concerns. For the valuation of both cohesion and separation of concerns two directions inside the traceability matrix of Figure 6 have to be considered.

Horizontal. The horizontal direction displays a number of marks for the realization of software categories though components or interfaces. For a high cohesion and well separated concerns, there should be categories realized only by components or interfaces that belong to one unit of design. In sum, *Application Server Calls*, *Data Queries*, *Data Validation*, *Dialog Lifecycle Actions*, *Dialog Navigation* and *Model Data Observer* are realized by several Quasar elements, and thus, different units of design. The first three categories are shared among the *ApplicationKernel* and *DialogKernel*. Thus, the resulting coupling between these design units will largely depend on the refinement of interfaces between both components.

Eventually, a mixture of *A* software categories can be a probable result when no *0* interfaces can be invented. The details of this client and server communication remain an open issue as well as the construction of data queries.

Besides, *Model Data Observer* is presented with two options that are either implemented by the *DialogKernel* (*DataRead*) or *Presentation* (*DataUpdate*). However, the complementary task of *Model Data Edit* is only briefly mentioned. Siedersleben states that the *Presentation* knew about the *DialogKernel* but not vice versa [4]. How the important task of changing dialog data is performed by the *Presentation* and what interfaces are required is left open.

 ISBN: 978-1-61208-367-4

Moreover, *Dialog Lifecycle Actions* are of less importance. They are rather stereotype operations that could be detailed by *0* type software. For the *Dialog Navigation*, there may be missing directions in the Quasar client reference architecture, so that responsibilities have to be refined on behalf of the developer. We wonder how dialog sequences resulting from task model specifications would affect the software category assignments. Maybe the Session cannot be marked as *0* software anymore, since it needs knowledge of the proper sequence of dialogs, which may finally be reused for different task model instances.

Vertical. A further assessment considers the vertical direction that reveals targets with many traceability-links. This can be a marker for lacking detail or even low cohesion. Those targets would take on too many responsibilities at once. There are multiple candidates that awake our attention.

As already stated above, the *ApplicationKernelService* needs further refinement, so that the way how calls and queries are performed by the *DialogKernel* are both detailed and differentiated concerning allowed data types and resulting coupling. Consequently, another major issue is the *DialogKernel* itself. This component is relatively vague in definition, so that tasks like calls to the *ApplicationKernel*, queries, the dialog data definition, data validation and the control of states need to be elaborated from scratch. Concerning functional requirements tracing, the *DialogKernel's* internal structure and state control are important issues that affect the resulting dependencies to requirements. For instance, it has to be decided what portions of a use case will be exclusively realized by the *Application Services* and what parts the *DialogKernel* is in charge of. Above all, the *DialogKernel* is likely to depend to some considerable extent on the *ApplicationKernel* and its *Domain Data Model*. In this regard, it has to be cleared how queries are to be handled from the *Dialog Data Model's* point of view. The *Dialog Data Model* can either be composed of pure entities, which may be embedded as interfaces or data transfer objects, or aggregations that are sourced from selected attributes of several entities retrieved by a query.

Furthermore, the *Presentation* also requires further elaboration in design. Being the complementary part of the *DialogKernel* in a dialog, the *Presentation* is declared as having its own data model in parallel to the *DialogKernel* in order to perform conversions to the *Technical Data Models*. The main data definition is assigned to the *DialogKernel*, since this component is in charge of any data retrieval from the *ApplicationKernel*. How the data related communication (read and edit) besides the notification of updates between *Presentation* and *DialogKernel* is originally intended remains another open issue. In this regard, design decisions on both interfaces and data types as well as their connection to the *Domain Data Model* have to be considered. Moreover, details about the triggering (*Presentation State Update*) and execution of *View State Changes* are missing. This is due to the unclear connection between *Presentation* and *DialogKernel*. When decisions about reactions on events are bound to *Presentation*, logical behavior will be closely coupled to views, so that they are less flexible for change and reuse. In addition, events can only be emitted by view elements and can not be triggered by the evaluation of gathered dialog data alone, since there is no link for the *DialogKernel* to initiate a *View State Change* via *Presentation State Update* when an event was forwarded.

Lastly, the *ViewDefinition* interface and related implementations inside the *Presentation* need more refinement. The coarse grained interface is employed for both handling view states and their initial construction. In this context, a developer would have to decide on how the *DialogKernel* may trigger the visual state changes as a result of its own states defined by *Dialog State Changes*.

3) Summary

Our review of the Quasar client revealed that this reference architecture is more advanced than common GUI patterns. Its main advantage lies in the division of *Controller* tasks among the *Presentation* and *DialogController*, so a better separation of concerns can be achieved. However, this results in increased complexity concerning the number and type of interfaces to be implemented.

In comparison to other architectural patterns, the Quasar client provides more detail and descriptions that give hints to many design decisions, but these are scattered among several sources [4][16][21][22] only available in German language. There was no comprehensive description published, which would provide every needed implementation detail. In the end the Quasar client remains vague with many important issues to solve by individual design decisions. Nevertheless, we learn from the traceability matrix in Figure 6 that there are already hints, which component is to take on what responsibility. In practice, this would yield only a partial improvement with respect to the common GUI patterns. In [1], Haft et al. state that the Quasar client could not be standardized, since most software projects required specific adaptations. The many individual refinements would affect the marking of software categories, so that the purity of them and the separation of concerns may not be maintained as intended. Even the Quasar client assumes that some portions of *AT* software cannot be avoided with conventional architectures relying on invasive frameworks.

To conclude, the Quasar architecture is not suitable for a straight forward implementation. As we see, there are still gaps in the reference architecture and the developer has to incorporate own thoughts in order reach the desired quality architecture. The separation of concerns can be improved with a customized Quasar client architecture, but this largely depends on the skills of the architect. In the end, the Quasar client may be a better, and foremost higher detailed, basis for reuse of architectural knowledge than the MVC variants.

V. RESULTS AND DISCUSSION

1) General Considerations

We derived a software category model that structures the dependencies among common responsibilities of GUI architecture design units. This set of categories can be of aid for the valuation of both the detail and separation of concerns of reference architectures or patterns. In the context of GUI design, the categories resemble different and delimited packages of knowledge, which are used to identify and map components. Later on, the dependencies among the categories will lead the design of interfaces between components [4] to achieve a minimum of coupling. Thus, the

proper distribution of identified categories among design units has an enormous impact on software quality.

Concerning the actual shape of the software categories tree, there might be different structures or aggregations possible (intermediate categories) but the final child elements clearly mark the occurring responsibilities. Currently, concerns like user profiles, additional assistance and authorization are not included. In general, terms in the field of GUI architecture are not used uniformly, so we rely on our category model that provides a clear description of tasks. Furthermore, the software categories may be adapted to fit other domains, since the separation of concerns is essential in most software architectures.

2) Major Issues in GUI Architecture Design

The available architectural patterns differ in structure as well as the encapsulation of concerns. Finally, there is no standardized GUI architecture ready for implementation. This is an issue here but also for mobile devices [26]. We analyzed the differences or missing details of presented architectural patterns and identified three major design issues that may have a considerable impact on GUI maintenance.

Firstly, a design decision has to treat the question what and how much application logic is being processed by a single dialog, or particularly its *DialogKernel*. Thus, the coordination and division of labor between dialog and application related components should clearly define what portions of the event processing chain will just be handled by the *DialogKernel*. As the primary controlling entity of a dialog, the *DialogKernel* acts as a client of the *ApplicationKernel* and its services [4][22]. The architect has to decide how much control flow will be implemented by the client and what operations or services are to be integrated in the controlling object's flow definition. For instance, the business logic can be separated by different layers like services, auxiliary services, domain model entities and data types [27]. The coordination of the various algorithms, which is essential to achieve the goals defined by use cases, can either be performed by the *ApplicationKernel* or the *DialogKernel* may govern the sequence of service calls and their combination. The so called orchestration of services to realize a certain use case is an option for the *DialogKernel*, since this design unit determines the data structure for user interaction. In this context, the *DialogKernel* directly can react to valid user inputs and may decide on the further processing via services or may even trigger corresponding state changes for the *Presentation*. How the latter is to be designed remains an open issue. Siedersleben states that the *ApplicationKernel* components constitute of use case realizations [4]. However, these components would definitely be incomplete use cases realizations, since the latter regularly require much user interaction. To conclude, the question arises how use case realizations are sub-divided among *ApplicationKernel* services (management of data structures and relationships), *DialogKernels* (logic for dialog flow and control of user interaction) and finally *Presentations* (visual part, in- and output UI-Controls). Ultimately, this design decision depends on the navigation structure and whether one *DialogKernel* may control a composition of *Presentation* units or sub-dialogs that form a complete dialog unit for the sake of one use case realization.

This leads us to the second issue that is concerned with the flow of dialog units or navigation among them. Recent research [28][29] investigated on the role of task models for structuring the flow of dialogs. In analogy to the above described issue of division of labor for use case realizations between *ApplicationKernel* and *DialogKernel*, the architect has to decide on the responsibilities of a single *DialogKernel* concerning the flow of dialogs. The question arises what part of the navigation is governed by higher situated components, e.g., a dedicated task controller, and what view changes are in the responsibility of the *DialogKernel*.

Thirdly, the Quasar software categories serve a main purpose to separate application from technical aspects, and thus, avoid *AT* software. As far as the GUI architecture is concerned, we identified two aspects where *AT* software does occur. The *Presentation* communicates with both the *GUI Framework* and *DialogKernel* in order to retrieve data inputs from the user. Eventually, the *Technical Data Models* of the *GUI Framework* and the *Dialog Data Model* have to be converted in the respective formats to enable information exchange. There may be a second conversion necessary between *Dialog Data Model* and *Domain Data Model* when the *DialogKernel* has to use a different data format. Another aspect of *AT* software is the transformation of the *Dialog Data Model* to visual representations, which are constructed by the *Presentation*. Accordingly, the *Presentation* needs to possess knowledge of both the proper selection, arrangement of UI-Controls and the usage, creation of the latter via the specific *GUI Framework* facilities. Besides the first two issues, these two *AT* software aspects can additionally increase maintenance efforts. To solve the third issue, conventional architectures will not suffice and specific designs for additional decoupling have to invented. An initial approach was formulated by Siedersleben and Denert in [16].

3) User Interface Patterns

Before we draw our conclusions, we briefly note how the incorporation of UIPs for the *Presentation* component may resolve the mixture of application and technical aspects. UIPs promise the reuse of visual layout and related interaction. The *Presentation* could be composed of these pattern units and would specify their contents via parameters. The UIP implementations would directly depend on the *GUI Framework* and no longer each *Presentation* unit. Therefore, fewer efforts would have to be spent on programming with *GUI Framework* facilities in the long run when UIPs could be reused extensively. The development could be focused on the *DialogKernel* design issues instead.

To integrate UIPs in the *Presentation*, the differentiated software categories for event processing will be of great value as they prepare the better adaptability and even exchange of *Presentation* units. Responsibilities would be centered in the *DialogKernel* to raise the flexibility of UIPs.

VI. Conclusion and Future Work

The scope of this work is a study of the prevailing issues of GUI architecture design. A software category tree on the basis of Quasar was elaborated, which displays common responsibilities for GUI architectures and their dependencies. With the aid of the software categories, we have analyzed the common GUI MVC pattern and the Quasar client reference

architecture. As result, we identified pattern specific and general issues of relevance for design decisions within GUI architecture development. The herein applied method with a decomposition of software categories and the tracing to an architecture model can be applied for other domains to assess the separation of concerns, cohesion and coupling.

Future work. The findings of this work will influence our further research into the implementation options for UIPs. The Quasar client proved to be the most advanced architecture publicly available. On the basis of the identified issues of that architecture, we will have to develop dedicated solutions to prepare a suitable target architecture for UIPs. We need to further assess the architecture variants outlined in our previous work [17]. The software categories will help us to plan and evaluate possible solutions. Whatever architecture variant will be favored, it definitely needs a software architecture of high quality with well separated concerns to accept UIPs as additional artifacts. The solution must resolve the identified GUI design issues to integrate UIPs in order to reduce the efforts for adaptation of GUIs.

REFERENCES

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

[2] T. Reenskaug, "Thing-Model-View-Editor. An example from a planningsystem," Xerox PARC technical note, 1979.05.12.

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

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

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

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

[7] M. Lindvall and K. Sandahl, "Practical implications of traceability," Software - Practice and Experience (SPE), vol. 26, issue 10, Oct. 1996, pp. 1161-1180.

[8] P. Mäder, O. Gotel, and I. Philippow, "Getting back to basics: promoting the use of a traceability information model in practice," The Fifth Workshop on Traceability in Emerging Forms of Software Engineering, IEEE, May 2009, pp. 21-25.

[9] J. Siedersleben, "An interfaced based architecture for business information systems," The Third International Workshop on Software Architecture (ISAW '98), ACM, Nov. 1998, pp. 125-128.

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

[11] 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.

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

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

[14] 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), IEEE Computer Society, July 1998, pp. 25-31.

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

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

[17] 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/.

[18] J. Dunkel and A. Holitschke, Softwarearchitektur für die Praxis [Software architecture for practice]. Berlin: Springer, 2003.

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

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

[21] J. Siedersleben (ed.), "Quasar: Die sd&m Standardarchitektur [Quasar: The standard architecture of sd&m]. Part 2, 2. edn. sd&m Research: 2003.

[22] M. Haft and B. Olleck, "Komponentenbasierte Client-Architektur [Component-based client architecture]," Informatik Spektrum, vol. 30, issue 3, June 2007, pp. 143-158, doi: 10.1007/s00287-007-0153-9.

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

[24] J. Cai, R. Kapila, and G. Pal, "HMVC: The layered pattern for developing strong client tiers," JavaWorld Magazine, http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html (2000), 2014.08.14.

[25] R. Stafford, "Service Layer," in [19].

[26] K. Sokolova, M. Lemercier, and L. Garcia, "Android passive MVC: a novel architecture model for the android application development," The Fifth International Conference on Pervasive Patterns and Applications (PATTERNS 2013), IARIA, May 27 - June 1 2013, pp 7-12.

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

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

[29] V. Tran, M. Kolp, J. Vanderdonckt, and Y. Wautelet, "Using task and data models for user interface declarative generation," The Twelfth International Conference on Enterprise Information Systems (ICEIS 2010), vol. 5, HCI, SciTePress, June 2010, pp. 155-160.

 ISBN: 978-1-61208-367-4