Modeling Responsibilities of Graphical User Interface Architectures via Software Categories

Stefan Wendler
Software Systems / Process Informatics Department
Ilmenau University of Technology
Ilmenau, Germany
stefan.wendler@tu-ilmenau.de

Abstract — **Business information of our days systems heavily rely on graphical user interfaces (GUIs) as a sub-system that provides rich interaction options to access business services and stands out with high usability. To develop and maintain a GUI sub-system, high efforts accumulate due to missing standard solutions and limited reuse of already established architectures. Published architectural patterns and few reference architectures are primary sources for GUI architecture development. However, these concepts need to be extensively adapted, since individual requirements are to be met and available sources do not describe all necessary details. These are fine-grained GUI responsibilities, differentiated state handling for application and presentation as well as implementation structures. Therefore, GUI development projects create high efforts and their resulting architecture often does not represent the desired separation of concerns, and so, is hard to maintain. These architectures are no proper foundation for the integration of recent user interface pattern (UIP) concepts, which promise a reuse of proven usability concepts and enable the automated generation of vast GUI parts. In this work, the design issues that occur during GUI architecture development are to be analyzed. To prepare the analysis, selected GUI architecture and pattern concepts are presented. Furthermore, the general responsibilities of GUI sub-systems and their structural elements are identified. In detail, software categories are applied to model the GUI responsibilities and their relationships by separating their concerns based on several dimensions of knowledge. The resulting software category tree serves as a basis to review the well-known model view controller pattern and the Quasar client architecture, which is a detailed GUI reference architecture of the domain. As result, the major design issues of GUI systems are derived and summarized. Eventually, the created GUI software category tree can be applied as a foundation for the creation, understanding and assessment of other GUI patterns or reference architectures.**

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 [2] for both development and later maintenance. According to a survey among 23 major Germany-based IT service companies, IT departments of banking, logistics and power supply industries, as well as medium-sized IT developers, the development efforts for GUI systems is still estimated to be considerably high compared to other common sub-systems of business information systems. On a basis of 100% total development effort, the aggregated efforts across all development phases were estimated for the four principal business information systems layers as follows: workflow layer 24,8%; presentation layer or GUI 26,8%; application layer 29,8% and lastly persistence layer 18,6%. In sum, the presentation layer was rated as the second highest concerning total effort.

The high efforts for GUI development apply for both standard and individual software systems as a high demand for individually designed GUI systems is actually present. The companies require their business information systems to be closely matched to their business processes. As a consequence, custom services are often to be developed or configured, which require a customized GUI to reflect the functional aspects. In addition, a high usability is almost always a desired goal to achieve during the development of new GUI dialogs.

Problems. However, GUI architectures are not standardized to the required detail, since historically applied patterns have not converged towards a detailed standard architecture that models every responsibility needed for considering current functional, usability and technological influences during development or maintenance. According to functional aspects, the higher degree of system integration into business processes demands for exact implementations of comprehensive requirement artifact types like use cases, tasks and business processes. The customers expect the GUI system to closely match the specified scenarios with dialogs that reflect the flow and branching of actions along with the proper display of context relevant and even optional data. Users no longer reenact those scenarios by activating the single functions with their belonging dialogs in the right order. They expect the GUI system to provide guidance instead, navigation facilities and adequate presentation layouts to attain a dialog structure that perfectly mirrors and complements the functional requirements specification.

Those current GUI development needs are facing rather old GUI architecture patterns like model view controller (MVC) [3] and its variants [4], which did not consider such a deep and vast requirements basis. To resolve some MVC limitations or add some detail, other MVC pattern derivates like HMVC [5], MVP [6][7], MVVM [8] and MVA, RMR, ADR (reference [9] provides some overview only) were

This work is an extensively revised and substantially augmented version of "A Software Category Model for Graphical User Interface Architectures", which appeared in the Proceedings of The Ninth International Conference on Software Engineering Advances (ICSEA 2014) [1].

introduced and occasionally found their role as a principal architecture of GUI frameworks. However, mostly single dialogs are considered by those patterns, so that concerns like the design of navigation among dialogs, the structuring and separation of visual layout, presentation control, dialog control [10] and application flow are not comprehensively described by a single pattern. There is no standard solution available by the books; many sources [11][12][13] focus on the proper handling of programming languages or mastering certain GUI framework facilities without paying much attention to architecture structuring. Thus, many details remain to be refined by the developers [14], who will adapt architectures individually for each system and most likely will not extract a commonly reusable architecture due to lacking time or budget.

According to Fowler [15], during the course of analyzing and refining patterns many different interpretations may emerge, so that there will be no common understanding of a single pattern and its involved roles. This may due to the complex structure of patterns, which are regularly containing several ideas at once that may even comprise smaller sub-patterns. Thus, developers will instantiate one pattern according to their gained understanding, experience with other patterns and the integration of surrounding frameworks and architecture aspects to be addressed.

Ultimately, there is no consensus on GUI patterns, which one offers the optimal structuring of responsibilities, so that it is fairly common to decide on their application and adaptation anew for each GUI development project. Although MVC is very commonly applied, this pattern also is very often misunderstood [15]. To apply common GUI architecture patterns in practice, several implementation problems have to be solved that are not sufficiently addressed by the patterns [10]. Besides, reference architectures [2][16] and several patterns (design and architectural) [17][18] had been suggested, but have not been properly integrated with traceability [19][20] concepts to keep track of requirements during architecture design.

Moreover, GUI frameworks often dictate to closely adopt a certain pattern-based architecture, so that they have a large impact on the GUI system's structure and often cannot be isolated properly to separate technical implementations from domain or project specific requirements.

So far, the functional aspects were considered. As far as the demand for high usability is concerned, the above mentioned patterns do not solve the integration of ergonomically effective presentation layouts or interaction designs. They focus on mere technical reuse of software architecture structure and do not consider content-based reuse.

Consequences. Foremost, GUI systems remain hard to develop concerning the effective adaptation of available patterns or reference architectures, as well as the cost-efficient implementation of current functional and usability requirements. In addition, developers may be frequently required to work with a certain GUI framework to be able to integrate the new created GUI system parts with an existing system or maintain a certain corporate design already in use with other neighboring systems. In the end, the coupling between system layers and the separation of concerns remain vague due to different pattern characteristics and project budgets.

Furthermore, 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 allocated to certain elements of the software architecture. These concerns are likely to be scattered among several units of design. Finally, the GUI and application sub-systems cannot be separated easily, so that the evolution of both depends on each other. Business logic tends to be scattered in the GUI dialogs [21] and the "smart UI antipattern" [22] may become a regular problem. Initially, the architecture was layered during design phase, but the encapsulation of components and separation of concerns did not prove in practice [21]. 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 [21] are still of relevance: standardized interfaces between layers are missing and technical frameworks dominate the architecture and evolution. Currently, there are often more than three layers in business information systems and the segregation got even more complex.

User interface patterns. There are perspectives that are promising to address the persisting issues. Current research is occupied with the integration of a new artifact type in the development of GUI systems. Being based on design pattern concepts and likewise description schemes [23], user interface patterns (UIPs) have been approached [24][25][26] 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 (HCI) patterns [27][28].

According to the generative nature of UIP integration approaches, 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. This new kind of pattern is intended to bridge the gap between descriptive HCI patterns and implementation oriented architecture patterns. Ultimately, with the application of UIPs the technical reuse of architecture structures of common design or architecture patterns shall be combined with the reuse of content relevant for ergonomics (visual design and layout, interaction design, HCI patterns) bound to certain design units, which usually remain abstract in common pattern descriptions. In that way, UIPs shall be stored in a repository to be configured and instantiated for different projects. In short, both technical architecture parts and visual design shall be coupled and reused in different contexts.

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. A commonly applicable GUI architecture has to be derived. 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 without the need to adapt other components. Otherwise, the previously described problems would persist: due to lacking standard solutions, each project would need an individual target architecture with every responsibility detailed to accept UIP instances. Generator templates would have to be created and revised over and over again for any GUI framework or platform. The visual designs of UIPs would only be available for specification and context configuration, but would miss a technical architecture for their implementation on a certain system. To be able to increase reuse with UIPs, a standardized architecture solution is truly needed. The individual refinement of patterns will greatly hinder the benefits UIP-based reuse would promise.

Whether UIPs will be generated, interpreted or provided by a virtual user interface [29][30] the resulting architecture will be at least as complex as for standard GUIs. Therefore, 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 first objective is to provide a detailed analysis of perpetual design problems. Design issues regularly occur whenever one of the following cases is encountered:

- Requirements are not met due to missing allocation of responsibilities to design units.
- Several design units share are certain set of responsibilities, so that either cohesion or separation of concerns is not ideal.
- One design unit takes responsibility for many tasks at once, and thus, may not represent a proper degree of cohesion.

Hence, we will have to identify the re-occurring responsibilities of GUI architectures to be able to analyze possible GUI design issues. In this regard, our second objective is to derive a pattern- and architecture-independent model of those responsibilities and their relationships.

On that basis, the frequently applied MVC pattern is reviewed. In addition, we will analyze the Quasar client reference architecture [2] that provides more detail than regular patterns and was created especially for the domain. Together with the presentation of selected related work, the responsibilities model and the analysis will lead us to reveal persisting issues in GUI design.

C. Structure of the Paper

The following section provides descriptions of common patterns and reference architecture considerations for GUIs of our particular domain. 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 and discussed 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 MVC pattern was introduced [3], which distinguishes three object types as abstractions to accept defined responsibilities. The typical roles of an MVC triad are the following: *View* and *Controller* comprise the GUI part; the *Model* represents the application parts with the core functionality and data structures [18].

With these roles, the MVC pattern promised a separation of concerns, modularity and even reuse of selected abstractions [31]. According to Fowler, one main idea of MVC was the concept of “Separated Presentation” [15][32]. Hereby, an application layer is separated from the GUI layer, which regularly accesses the former but not vice versa. In other words, the GUI part of a system strictly represents an independently developed sub-system, comprised of *View* and *Controller* elements, that calls the application or domain layer services by using a dedicated interface element provided by the *Model* of the MVC triad. Thus, the communication with the application layer is mostly initiated and controlled by the GUI part of a system. However, the application layer does call the GUI layer in a clearly defined way: by applying the observer pattern [18][17] *Views* are promptly updated whenever changes to the application layer or *Model* part are committed. This design allows for multiple *Views* sharing a certain *Model* and displaying different data in different ways.

In Figure 1, we present a possible architecture application diagram of the classic MVC pattern. Please note that an interface notation was used to describe the visibility (a certain set of operations) each involved design unit has on its interaction partners.

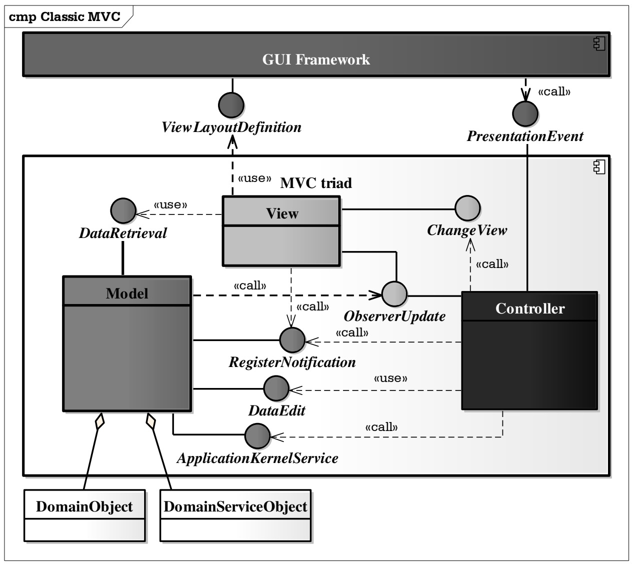


Figure 1. The classic MVC architecture pattern described by the three roles *Model*, *View* and *Controller* and their typical interfaces.

The initial setup of the triad is supported by the interfaces *ViewLayoutDefinition* (creation of screen layout, definition of UI-Controls) and *RegisterNotification*, which enables both *Views* and *Controllers* to receive notifications whenever *Model* data has changed. So, the latter is part of the observer pattern implementation. It should be considered that in the original MVC design applied in Smalltalk environments the access to the *Model* was strictly differentiated among *View* and *Controller*: the read-only *DataRetrieval* interface is used by the *View* to update its UI-Controls with current data whenever changes to *Model* have been applied. The retrieval of data by the *View* is typically preceded by a call from the *Model* via *ObserverUpdate*. In contrast, the *DataEdit* is a write-operation interface exclusively called by the *Controller* to apply changes to the *Model*, e.g., when the user has entered new data during interaction with the *View's* UI-Controls. Typical results that follow a user interaction scenario from the *Controller's* perspective are the previously mentioned change of *Model's* data (*DataEdit*), a request to the *View* to alter its display (*ChangeView*), and finally, a value creating call to the *Model* (*ApplicationKernelService*) that processes application data and changes the system's state concerning business data.

Besides these elementary interfaces and basic interaction mechanisms, the MVC pattern is affected by several problems.

Firstly, there exist many sources of the MVC pattern, which either do not cover the pattern with its multiple facets in entirety or are more or less influenced by the specific requirements of an application environment like certain GUI frameworks for either desktop or web clients. We consulted references [15], [31], [32], [33], [34] and [35] for related work. In addition, a widely accepted and often cited description can be found in [18], which was considered here of course. As mentioned in the introduction, there is no consensus on GUI patterns and their details. Ultimately, MVC ends up as "the most misquoted" pattern [15].

Secondly, the classic MVC pattern is bound to the Smalltalk environment and its basic facilities like abstract classes to create each specific member of the triad by using inheritance. As a result, the complete application was woven in MVC as a principal architecture or architectural style. This is often not applicable for nowadays system layers and current GUI frameworks. The classic MVC is conflicting and must be adapted to modern needs. For instance, Karagkasidis [10] discussed some implementation variants for a Java based MVC design.

Thirdly, from a practical point of view the classic MVC pattern misses many details that are essential to enable its benefits of modularity and separated concerns. Karagkasidis [10] already provided an elaborate examination of different concerns among popular GUI architecture patterns including MVC. In sum, the creation and assembly of GUI layout, user event handling, dialog control and the integration with business logic were identified as topics with several implementation issues.

In this regard, the MVC 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: a *Controller* has to handle the technical events (*PresentationEvent*), update the data of the *Model* (*DataEdit*), delegate the *View* to adapt its layout (*ChangeView*) to current data state, and finally, initiate the concluding processing of *Model* data by the application kernel (*ApplicationKernelService*). 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 *DataRetrieval*.

To cope with the close coupling and missing details, several variations of the MVC have been discussed [4][10]. In general, the variations in design differ concerning the distribution of responsibilities among the three abstractions. Several more patterns [6][7][18][32] 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. Their effectiveness can hardly be evaluated for the long term maintenance or a standardization perspective, since there are no elaborate or comprehensive descriptions available; some MVC derivates are only sourced from websites or weblogs [9][36]; comprehensive accounts on MVC variations are still under construction [37] or do not cover all variations.

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 the event processing within GUI architectures as described by Siedersleben [38]. In Figure 2, a variation of this model is displayed. Thereby, technical events will be emitted from the *Operating 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 and the associations between components therein.

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 *Dialog* 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 the task to communicate with the *ApplicationKernel* and its components instead.

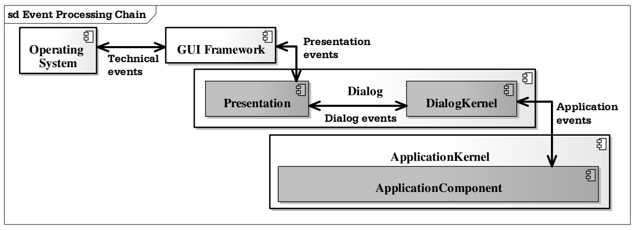


Figure 2. Value creation chain of graphical user interfaces derived from [38].

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*. Thereby, the *Dialogs* fulfill the purpose to connect the users with the main business services provided by *ApplicationComponents*. First of all, several user inputs that result in events need to be enhanced with further information. Then they can finally be forwarded through the components to trigger business services, which create business value. That is why Siedersleben speaks of a "value creation chain" [16][38].

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 [29]. 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 [29] 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) [16]. 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 [2] and the object-relational mapper Quasar Persistence [39] have been published. Conversely, the main ideas of standardization were neglected in [2] and reference architecture elements should fill the gap.

Software categories. As far as the component identification is concerned, so called software categories [16] were introduced. They consist of the five categories *0*, *A*, *T*, *R* and *AT*.

0 software 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. So, *A* software would represent the core of each business information system.

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 [16]. 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 [2][16] has to be mentioned for the scope of our work. Compared to common GUI architecture patterns, the Quasar GUI client architecture resembles a comprehensive architecture addressing the specific needs of a domain by incorporation of pattern elements and certain refinements.

The main parts of that architecture are illustrated by Figure 3 that is derived from [16], since this is the most detailed source available. The interface names in brackets quote the original but not very descriptive designations. The unique elements of the Quasar client architecture are the following three aspects:

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.

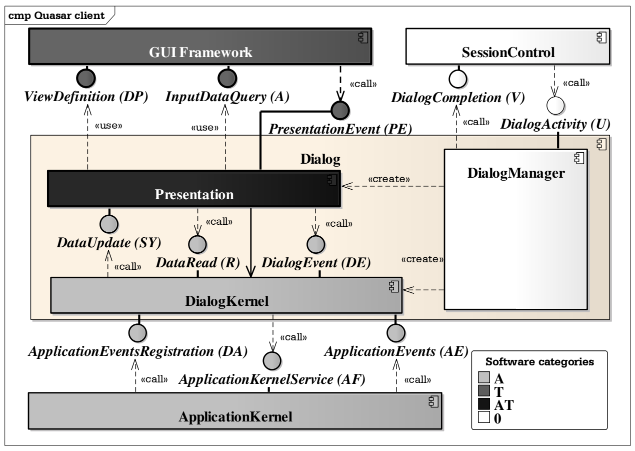


Figure 3. The Quasar client architecture based on [16].

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*. To create 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 [2].

III. General GUI Responsibilities Model

A. *Problem Statement*

As we learned from the introduction, standardized GUI architectures are not available, so that custom architectures prevail. Accessible architecture sources remain only as references to be adapted to specific requirements besides standardization efforts. The basic GUI patterns and the more detailed Quasar client reference architecture are too abstract and general to describe detailed responsibilities required for implementation purposes. Hence, our conclusion from Section II is that developers have to select and always adapt a MVC or other GUI architecture pattern variant suitable for their domain.

Although the available sources will not model an extensive set of GUI responsibilities, they provide a basic set of tasks and associated components. A closer examination of given sources proved that they may complement each other, as some sources are more focused on certain responsibilities than others. A common intersection of responsibilities can easily be found. However, it is challenging to enhance this intersection in order to obtain an almost complete set of GUI responsibilities.

Finally, those GUI responsibilities have to be modeled in a systematic way, but independently from any specific pattern or framework. The target architecture for UIPs has yet to be created and it would be of little use to modify existing architectures without having identified the prevailing design issues. In addition, the influence of UIPs on the target architecture and these issues can only be understood when a complete set of GUI responsibilities was identified.

The software categories of Quasar, which were introduced in Section II.C, can serve the purpose of modeling the GUI responsibilities, since they were invented to model the occurring concerns of a system's architecture prior to the identification of components. In the following section, we will review this concept.

B. *Quasar Software Categories Reviewed*

We found that 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.

Relationships. 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 (*0*), domain (*A*) and technical related (*T*) as well as mixed domain and technical (*AT*) concepts. Figure 4 displays those basic categories and their relationships. The dependencies in Figure 4 symbolize, which specialized category is derived from a more basic one. In this regard, *0* software is the parent category to be used for the composition of every other category. The elementary data structures and operations of *0* software are used to form other and often more complex data structures with their specialized operations that are unique in their purpose, which designates their final categorization.

Refinement. The further and project relevant refinement of the basic categories *A* and *T* 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. During refinement each category will symbolize a certain concern of system. In this regard, "concerns" represent heavily abstracted requirements and related functions. Siedersleben [16] 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.

Complexity. By refining parent categories, a number of child categories are created that directly depend on each parent category and implicitly take over the dependencies of their parents. Following that way, it is ensured that every category may access the basic programming language facilities modeled by the *0* software category. Moreover, Siedersleben [16] speaks of complexity when refinements are created. It is obvious that refined categories truly create more complex units of design, since they potentially contain or access their own knowledge with the addition of all ancestor parent categories.

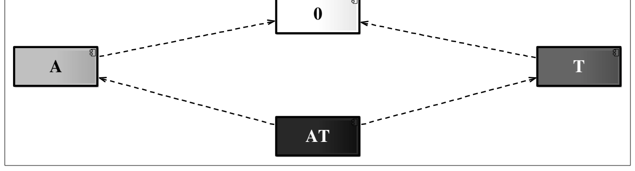


Figure 4. The basic software categories of Quasar [16].

Traceability. On that basis, refined software categories can be used to judge the purity of traceability-link [19][20] targets, meaning that these 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 [20]. Thus, the general or refined responsibilities of design elements will be visible, so traceability-link targets can be more detailed and better differentiated.

Problems. 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 [16][21], this might either be specialized knowledge how to handle certain algorithms and data structures or dependencies of an entity.

Moreover, there are only few examples [16] that explain the proper usage of the software categories. The related sources about Quasar [2][38][40][41] only use the basic software categories to mark components, but do not establish a category tree with refinements like this is done for a card game in [16].

Lastly, there exists no standard modeling concept for the software categories of Quasar. This issue could be regarded as a problem in analogy how to model architectures or identify classes. One could imply that the software categories miss comparative hierarchical concepts for their modeling like they are available for common design of architectures: architectural styles drive the identification of components. The inner design of greater components can be guided by selected architectural patterns (MVC can be given as an example). Consequently, the patterns with their defined roles drive the identification of classes and the latter serve to instantiate needed objects. However, nothing comparative is mentioned by available sources about the software categories of Quasar.

In sum, missing aspects for software category modeling are the following:

- Software category definition and delimitation,
- Software category identification approach,
- Software category standard modeling levels or style, arrangement for ease of readability or understanding

In the next sub-section, we will try to resolve these issues of software category modeling as far as our gathered knowledge and experience on this concept will guide us.

C. Rationale on Software Category Modeling

In this section, we will have to cope with the previously described problems of the software category modeling. We will have to find a way how a fine-grained responsibilities model based on the software category instrument suggested by Quasar can be established.

1) Software Category Modeling Purpose

The software categories are intended to refine tasks and document gaps left open by the available patterns. According to the Quasar rules and ideals [16], the category model to be created will represent a model with least coupling and cohesive elements that allows for planning dependencies among potential units of design. The categorization will be used in analogy to the suggested identification of components [16]; this step is essential to maintain separated concerns between identified responsibilities. Thus, the found responsibilities can be re-allocated during the development of a target architecture for UIPs or for solving common GUI design issues, but their separation can be maintained for a gain in software architecture quality and interface design with least coupling.

2) Software Category Modeling Levels

Quasar examples. With the given explanations, the software categories' scope remains vague. Therefore, we analyzed the provided example software category trees in reference [16]: on the one hand, some trees model abstract concepts like GUI, Swing and data access. On the other hand, the categories are applied to express certain component instances of a particular sub-system, as this is shown for an application kernel component dedicated to services a book library would offer.

From these examples, we conclude that a category tree can be situated on two principal levels of refinement: a software category tree that models abstract concepts and a tree, which is used to represent certain instances of a chosen concept of the former, are to be differentiated.

Abstract concept tree. The abstract description level is used to identify the general areas of knowledge that occur in a system and its components. This category tree is an abstract view on responsibilities that we understand as the arrangement of meta-types, which are permitted to occur in a system. So, the software categories on that level determine what type of tasks or sets of responsibilities are to be considered. Each set of responsibilities will correspond to a certain component stereotype. We understand that level of modeling in analogy to the object-oriented (OO) class concept: software categories model meta-types for design units to be identified. As OO classes determine what kind of objects can be instantiated, the software categories establish the types of design units, which define the software architecture's structural components.

The software categories of the abstract level are derived from the two basic categories *A* and *T*, and thus, the fundamental areas of knowledge of domain specific logic and technical interactions within the software architecture. Figure 5 illustrates an example for an abstract software category tree and its meta-types. Each meta-type expresses a set of tasks or responsibilities like this is the case for categories like *GUI dialog component*, *Application kernel component* and *File system persistence*, which express that layers or even components fulfilling the general task of proving application logic, a graphical user interface and file system based access will be present in the system. This kind of modeling of software categories can be understood as principal or general architecture modeling where the required layers and major component types are identified. In other words, the abstract software category tree answers the following question: what layer, component or other types of design units do occur in a system?

The sub-categories of the meta-types will be the actual layers, components, classes, and operations depending on the chosen detail in the hierarchy of modeling. According to traceability concepts mentioned in the previous section, the meta-types symbolize traceability-link targets in a taceability-metamodel: these are the principally allowed target types. For instance, a primary and simple distinction based on Figure 5 can be made between application and GUI components. So, requirements can either be associated to one of each type. This distinction is rather simple, but more effective than just allow the requirements to be traced to any arbitrary design entity. Another example can be derived from Evans' [22] domain model stereotypes. He identified concepts like services, entities, factories and value objects. These are abstract, but more concrete than arbitrary design units, and could be modeled in a software category tree as meta-types. Any other pattern type that has distinctive roles and their tasks described could be modeled with an abstract software category tree. In this regard, patterns with their set of characteristic classes can fill the gap that exists between components and bare classes: with the aid of software categories they permit the modeling of collaborations.

The sole modeling of one pattern makes little sense as the pattern's own description would suffice and most likely would be more detailed rather than a corresponding software categories tree. However, the modeling of system specific meta-types and the integration of patterns could be beneficial. Thereby, the categories would express the sum of all potential instances and the fact, that a certain pattern is present at a certain level in the systems's hierarchy of needed or allowed software categories. In addition, the software categories could be used to arrange a certain pattern and its roles in order with the existing hierarchy of design units. As result, the single roles or elements of a pattern do not need to be allocated to a fixed design hierarchy level like OO classes; they could be assigned to components as well.

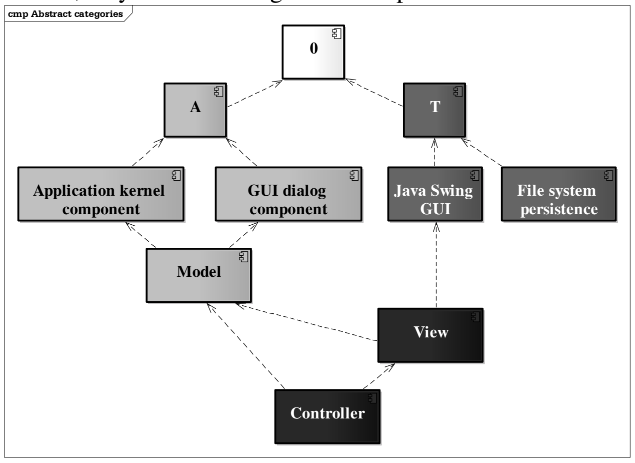


Figure 5. An example of an abstract software category tree.

This approach could be used for the refinement or even the combination of several patterns to structure a custom hierarchy or collaboration of classes.

For the sake of the example, Figure 5 was detailed with the categories *Model*, *View* and *Controller* to express that the MVC pattern (see Section II.A) will be applied in this system. In addition, the influences for that specific pattern application were added as dependencies among the software categories.

Accordingly, the *View* will be determined both by knowledge how to build visual forms with *Java Swing GUI* and how to properly access (assignment of data to UI-Controls in the correct order) the business data provided by the *Model*. In addition, *View* is implicitly determined by knowledge about the system's GUI specification with required layouts for certain functions or use cases represented by the *GUI dialog component* category. By maintaining the dependency to the *Model*, the *View* implicitly is connected to the parent categories including *GUI dialog component* on higher levels up to the basic *0* category, which is needed for the realization of every software category. Moreover, the *Controller* category is both influenced by the *Model* and *View* category: to perform data changes and coordinate application service calls, the dependency or knowledge of the *Controller* category must span the *Model* internals. The dependency on the *View* expresses that the *Controller* has to know about the *View's* structure or state to be able to request a proper change of the current screen layout or react on a certain UI-Control event trigger. The knowledge on *Java Swing GUI*, which is required for the *Controller* to be able to implement GUI framework specific event listener interfaces, is incorporated implicitly with the dependency on the *View* category.

However, this example points out what difficulties may occur by the integration of GUI or architectural patterns in a custom component architecture. Foremost, the three categories *Model*, *View* and *Controller* symbolize rather abstract concepts as they are described by the sources mentioned in Section II.A. More details about these three stereotypes have to be revealed in order to prepare the derivation of system specific instances and their implementation. Therein the difficulties are situated, whilst there is no consensus about the further refinement of each category or pattern role. Since acclaimed sources [3][18][32] do not provide sufficient details for current requirements, several different refinements [4][10] or interpretations for the MVC exist that result in varying dependencies and may differ from our example in Figure 5. Thus, the inner structure of each MVC category is not clearly determined and may vary as well. So will be the final quality of architecture and the separation of concerns depended on individual refinements. We could further detail each MVC category to achieve a clear distribution of responsibilities and guide the identification of smaller design units such as interfaces, classes and operations. This step can be quite helpful, since components are the ordinary corresponding unit of design for software categories [16], but these units are to be assigned to available programming language elements. Common programming languages do not feature a component as a unit for implementation after all.

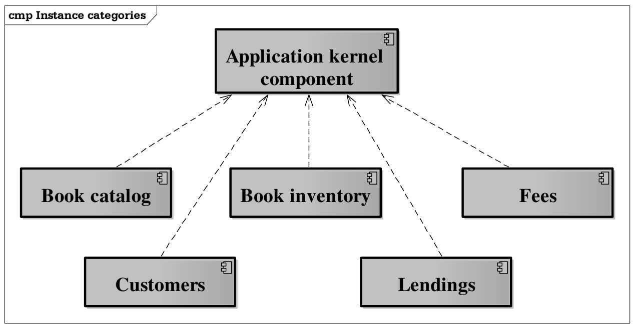


Figure 6. An example software category tree derived from [16] displaying identified components on the basis of a meta-type software category.

Refined software categories on the basis of certain class collaborations provide a modeling level in between and may fill the gap.

Instance tree. A second modeling level of software categories can be applied on the basis of the abstract concept tree. When the meta-types have been identified, the system specific instances or actual components or even classes need to be identified based on the found categories.

For instance, a software project would need 20 dialogs to appear in a system, which would contain 30 *View* instances, since 10 dialogs each would require two separate *Views*. These categories with their scope set to instances resemble concrete traceability-link targets in a project that are part of certain associations or dependencies.

Figure 6 displays an example based on Figure 5 where the abstract meta-type category *Application kernel component* was detailed with five needed instances as sub-categories. One could insert a suitable pattern for *Application kernel component* in Figure 5 like this was done for the MVC. Maybe Evans' domain concepts [22] could detail the *Application kernel component* as mentioned above, but this would alter the level of detail of Figure 6 as well.

It is obvious that the relationships of Figure 6 are rather simple and stereotype. We are inclined that the instance categories may introduce relationships among each other and eventually alter the dependencies inherited from the abstract parent software category. But these considerations are out of the scope of this work.

Summary. We outlined how the software categories of Quasar can be used to describe patterns in more detail or independently describe their responsibilities. We will tend to the described pattern refinement problem, and so, follow a similar way as seen in Figure 5. Our idea is to compose the GUI responsibilities from several sources at once and make use of an abstract software category tree to arrange them in an appropriate way. So, the categories will serve as the means for structuring, grouping and proper separation of responsibilities.

3) *Software Category Identification Approach*

We seek to establish a basis for the responsibilities that are regularly discovered in a GUI architecture. Our approach is depicted in Figure 7.

In detail, we will rely on four different kinds of sources and analyze them to identify the GUI responsibilities:

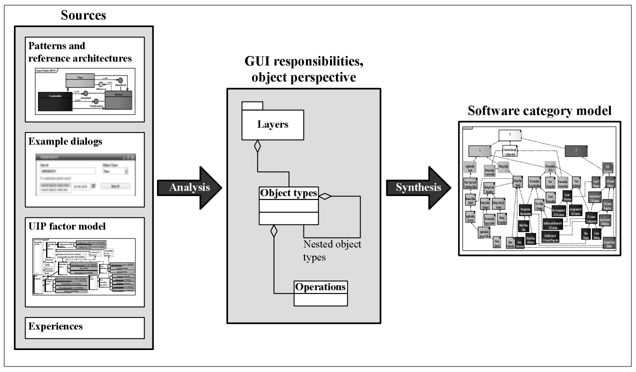


Figure 7. Process applied for the derivation of GUI software categories.

relevant responsibilities will be derived mainly from related work about patterns and reference architectures; considered sources are references [3], [4], [5], [6], [7], [8], [10], [14], [15], [16], [18], [29], [31], [32], [33], [34], [35], [36], [37] and [38]. In addition, we rely on more sources not described in this work: we use sample dialogs, consult the UIP factor model [42] and integrate own experiences from software development projects.

In fact, we do a decomposition of GUI architectures to rather atomic object types, related operations and nested object types. These entities 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 (*T* software). Finally, the found responsibilities will be assigned to individual software categories, which will be used according to the rules of Quasar to synthesize an abstract software category tree displaying GUI architecture responsibility concepts.

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. With that step, the main ordering concept of the analysis in the middle column of Figure 7 is established. Finally, the main layer categories *Application Kernel*, *Dialog Logic*, *Presentation* were identified as *A* category children, since they depend on the individual domain-specific requirements of a software system. Moreover, *Presentation* and *Dialog Logic* were separated as software categories according to the event processing chain of Figure 2.

Category identification. As the tree gets more detailed, software categories will become very fine grained and embody components, collaborations, 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.

To identify each of the refined categories, we applied several rules of thumb. During the analysis of GUI architectures and patterns, we derived categories from the different families of operations that regularly occur in the scope of certain units of design. In general, these were the definition or modification of object types or their data types, event triggering or processing, as well as forwarding of both data and events. These kinds of operations occur for different layers like technical or application related objects of general GUI pattern components that are common for MVC or the Quasar client. The different layers symbolize certain levels in the software category tree and were derived from reasonable abstractions like application logic, presentation logic and presentation technology. These layers should help us to prepare a coarse-grained order principle of GUI responsibilities and let us establish a high level categorization. The applied layers are partially related to the ones outlined in [10]. We alter and extend the given description. The layers will be explained in the following listing:

- *Application logic*: The objects and operations are dedicated to realize the core functional requirements of a business information system.
- *Presentation logic*: This layer is settled in-between the two other layers. So, it resumes tasks that cannot clearly be assigned to one of the other layers. These tasks include the handling of states that affect the visual appearance and navigation among different screens or dialogs. Furthermore, the logic that determines what application logic calls are appropriate in a certain state or how data states influence the screen layout and its UI-Control states is realizes in that layer. In sum, it couples application logic and presentation technology layers to create a seamless flow of interaction. This is done by translation of events emitted from presentation technology to application logic services and data changes. Changed data has to be reflected on the presentation technology layer; hence the presentation logic has to initiate a respective update of the presentation technology layer. Basically, this layer addresses the need that the different components on the various layers do influence each other with their internal state changes as this is described by Karagkasidis [10].
- *Presentation technology*: Both GUI framework and system objects are combined to create or alter the views and visual effects of a GUI for displaying data and interaction facilities. The visual states are implemented with that layer's objects and operations, but its tasks do not include the logic required for deciding what state is appropriate in a certain situation. In addition, the reaction to user inputs and the activation of event processing are further tasks of that layer.

For each of the layers, 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* or *A* 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 software category would be mixed, if the imports will demand for the addition of types defined exclusively in the imported unit of design.

These considerations led us to finding software categories on different levels of an abstract software category tree; it also inspired us to establish a clear definition of software categories that is presented in the following sub-section.

4) *Software Category Definition and Delimitation*

So far, the fundamentals surrounding software categories were described. It is still to be declared what are the concrete contents of a software category. This aspect is essential to describe each software category's individual details and to be able to delimit them.

However, the clarification of software category contents is not supported by available sources. Therefore, we derived certain dimensions that exist in a hierarchical order of dependency. These dimensions outline the contents of one specific software category. Figure 8 illustrates what dimensions define the knowledge that resides inside software categories. The following paragraphs will explain each of the dimensions.

Specific content. Each software category is dedicated to a specific topic or area of knowledge. All sub-ordinate dimensions depend on the choice of that content. Thus, the dimension acts as a filter to permit the inclusion or exclusion of certain *Contained entities,* what *Type of operation* is to be performed with them, and finally, what *Knowledge* must be possessed for the implementation or usage of defined operations.

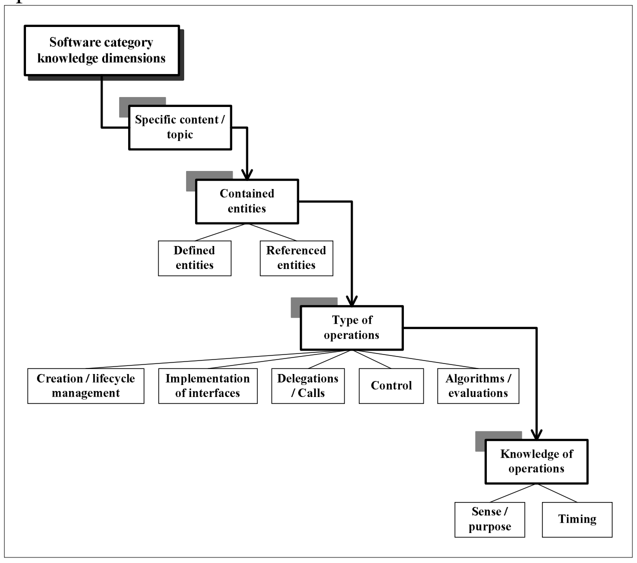


Figure 8. Software category definition via successive dimensions.

For instance, the software category *Java Swing GUI* of Figure 5 permits the containment of every class of the Java Swing GUI framework and all basic Java foundation classes that can be assigned to *0* Software further upwards in the tree.

The expression of that dimension can hardly be formal. A semi-formal approach may be established by assigning certain requirement models as the specific content.

Contained entities. This dimension determines what object types or units of design are to be considered inside the software category. Two particular cases are to be differentiated: in the first case, a software category may introduce and define specific units of design. These originate from and exist in the scope of that particular software category. In the other case, the software category is referencing entities or units of design that originate from other software categories and are not defined inside the current software category. This case often occurs for the import of interfaces connecting different components or classes or for the incorporation of foreign data structures.

The entities may consist of layers, components, interfaces, classes or even smaller units of design. It largely depends on the hierarchy level the software category resides on. To constrain the set of entities the first dimension puts up a global limitation for the software category. The scope of topics, and so, the number of contained entities differs greatly with respect to the given software category hierarchy level. Thus, the concept of software categories follows a hierarchical de-composition downwards the tree. It is of the essence for each architect to find a suitable level for detailed modeling to achieve proper cohesion and no coarse or too fine-grained units of design.

Besides, the second dimension affects the third dimension in a way that objects and data structures both for calling (allowed parameters) and implementing (interfaces and data structures) operations are defined. According to the refinement of software categories, the dependencies of the current software category express that all *Contained entities* from the previously defined parent categories are implicitly contained as *Referenced entities*.

With the given definition of the second dimension a software category may formally be defined by the entities it contains or references.

Type of operations. The next dimension is concerned about the general type of operations performed with the previously *Contained entities*. There are various options:

- *Creation*: Entities are created with the knowledge of the software category. Additionally, the entire lifecycle of entities may be governed.
- *Implementation*: Interfaces required to interact with certain entities are implemented. These can be call-back interfaces that are typical for the event listeners of GUI frameworks. Furthermore, interfaces can be defined by superior entities that need a certain set of operations to be implemented by lower situated interaction partners.
- *Calls / delegations*: Operations of other entities are called and the control is passed on to them.
- *Control*: Other entities are called with their operations but the control remains inside the software category. This kind of operation is applied in order to coordinate a flow of operations or events.
- *Algorithms*: Domain specific calculations are performed or technical routines activated. The results are obtained from the knowledge present in the software category or are gathered from *Referenced entities* operations that may eventually be used for enhancement or aggregation.

Depending on the type of operation combined with the considered entities, the software category type, its purity or coloring may change. For instance, the *Controller* of Figure 5 is no pure category, but of a mixed type, since it controls both the appearance of the *View* (compare *ChangeView* interface of Figure 1) and simultaneously coordinates calls to the Application kernel components (compare *ApplicationKernelService* interface of Figure 1). So it must possess knowledge about both topics at once. In addition, the *Controller* has to implement event call-back interfaces that are referenced within its scope but are defined in and constitute parts of other software categories like *Java Swing GUI*.

Both the second and third dimension can be sharply determined and delimited by enumeration of entities and operation types performed with them. Therefore, the two dimensions together represent the formal part of a software category's definition.

Knowlegde of operations. This final dimension expresses the proper moment in time and purpose of a contained operation inside the software category. Essentially, it represents the proper sequence, atomic steps and meaning of operations. This knowledge combined with the definitions of the previous dimensions embodies the ability to finally implement the operations of a software category.

D. Graphical User Interface Software Category Model

In this section, we will apply the approach presented with Figure 7 before we describe the GUI software category tree. Hence, the following sub-sections will analyze the responsibilities covered by GUI architectural patterns and their sources introduced in Section II. We will begin with the MVC and its variants, which is followed by the analysis of the Quasar client.

1) Analysis of the Model View Controller Responsibilities

The responsibilities described by the MVC pattern and its variants are summarized in TABLE I. Please note that the sources for the different MVC responsibilities are not completely mentioned; only the primary or sources with significant descriptions of these responsibilities are considered. Moreover, the assignments of operations may vary due to several MVC design options, which are exemplarily described in [4] and [10]. Our scope is the completeness of responsibilities and not the display of different design options.

2) Analysis of the Quasar Client Reference Architecture Responsibilities

Compared to the previously illustrated MVC responsibilities, the Quasar client includes many of these but considerably adds detail concerning the presentation logic and application logic layers. According to Siedersleben [16], the Quasar client compares to MVC as follows: the *View* is contained in the *Presentation*.

TABLE I. MODEL VIEW CONTROLLER PATTERN RESPONSIBILITIES.

Pattern role	Responsibility	Operations	Defined / referenced entities	Layer
Model	stores business data [18][31], provides results of data queries or intermediary object data [31]	read model data, change model data	Defined / referenced: data read and write interfaces for business objects and data types, aggregates or selections of business objects and their attributes (intermediaries [31]) (Inclusion or references depend on the realization of the model as a part of the application / business layer or as a separate unit of design.)	application logic
	validate data [4][35], provide additional information for visual interpretation of data [15]	validate data, read data interpretation information	Defined: data interpretation information Referenced: business data types and validation information	application logic
	provide an interface for calling application services [18]	call application service	Referenced: application services, business objects and data types as parameters	application logic
	register observers to be notified upon data changes [18]	register observer, deregister observer	Defined: list of observers Referenced: observer interface	presentation logic
	notify observers about data changes [18][31]	notify observers	Defined: setChanged interface Referenced: observer interface	presentation logic
View	display data, information and functions [18][31], arrange screen layout [31][15], visually interpret data [15], highlight validation errors	display initial screen, change screen layout, read model data, interpret model data	Defined: possibly specializations of GUI framework classes (may be used for data interpretation) Referenced: UI-Controls and layout managing facilities provided by the GUI framework, model data	presentation technology
	update data display [18]	read model data, update UI-Controls	Defined: update display interface Referenced: UI-Controls provided by the GUI framework, model data	presentation technology
	transform business data to technical GUI data model [31][35][16]	read model data, transform data	Referenced: model data, UI-Control data models required by the GUI framework	application logic, presentation technology
	create corresponding controller [18]	create controller	Referenced: associated controller	presentation logic
	register as observer of the model [18]	register observer, deregister observer	Referenced: model observer interface	presentation logic
	composition of hierarchical views [18][10]	create sub-view	Referenced: subordinate views, UI-Controls provided by the GUI framework	presentation technology
Controller	receive and react to user input [18][31]	handle event	Referenced: event listener interface provided by the GUI framework, possibly view's UI-Controls to determine event source and react differentiated	presentation technology
	translate events to service requests of either model or view [18][31]	call model service, change model data, call view display update	Referenced: model service interface, model data interface, view state change interface	presentation logic, presentation technology
	register as observer of the model [18]	register observer, deregister observer	Referenced: model observer interface	presentation logic
	update upon receiving notification from model [18]	update controller state, update view state	Defined: update controller interface Referenced: view state change interface, model data	presentation technology, presentation logic

TABLE II. QUASAR CLIENT REFERENCE ARCHITECTURE RESPONSIBILITIES.

Pattern role	Responsibility	Operations	Defined / referenced entities	Layer
Presentation	display data, information and screen layout, provide a proper localization	display initial screen, change screen layout (DP), read dialog data (R)	Defined: possibly specializations of GUI framework classes (may be used for data interpretation), presentation data model Referenced: UI-Controls and layout managing facilities provided by the GUI framework, dialog data model, localization data	presentation technology
	react to user input	handle presentation event (PE)	Referenced: event listener interface provided by the GUI framework	presentation technology
	forward events to dialog kernel when events are out of presentations' scope, attach event data	forward dialog event (DE), forward event data	Referenced: dialog event interface, presentation data model	presentation logic
	update upon receiving notification from dialog kernel	update presentation state (SY)	Defined: presentation data model, update presentation state interface Referenced: UI-Controls provided by the GUI framework, dialog data model	presentation technology, presentation logic
	control presentation states and trigger changes in screen display	change presentation state	Defined: presentation states Referenced: UI-Controls and layout state, presentation data model	presentation technology, presentation logic
	transform dialog data to presentation data	read dialog data (R), transform data	Defined: presentation data model Referenced: dialog data model, UI-Control data models required by the GUI framework	application logic, presentation technology
	validate input data to ensure proper formats are entered by the user	validate presentation data	Defined: presentation data model Referenced: business data types and validation information	application logic
Dialog kernel	handle dialog events, control dialog states and dialog lifecycle	forward dialog event (DE), change dialog states, close dialog, open sub-dialog	Defined: dialog states model Referenced: sub-dialogs	presentation logic
	control data states and retrieve data from the application kernel	ApplicationKernelService (AF), update dialog data model, update dialog state	Defined: dialog data model Referenced: application data model, application data queries	presentation logic, application logic
	notify presentation about data changes	update presentation state (SY)	Referenced: update presentation state interface	presentation logic
	translate events to service requests for the application kernel	ApplicationKernelService (AF)	Defined: dialog data model Referenced: application kernel service interface, application data model	application logic
	update upon receiving notification from application kernel	ApplicationEvents (AE), update dialog state, update dialog data model	Defined: dialog data model, dialog states model, ApplicationEvents interface	application logic, presentation logic
	register application kernel as observers of data or state changes	ApplicationEventsRegistration (DA)	Defined: ApplicationEventsRegistration interface Referenced: ApplicationKernelService interface, list of observers	application logic
	validate dialog data before calling application kernel services	validate dialog data	Defined: dialog data model Referenced: application data model, business data types and validation information	application logic
Dialog manager	control the lifecycle of the dialog composition	create and close dialog kernel, create and close presentation	Referenced: associated dialog kernel and presentation	presentation logic

Controller tasks are shared among *Presentation* and *DialogKernel*; they implement different control facilities with respect to their individual scopes (presentation technology and presentation logic). Lastly, the *Model* is realized by the data models of *Presentation* and *DialogKernel*.

In TABLE II, the responsibilities of the Quasar client, which we could reveal from references [2][14][16], are presented. Please note that Siedersleben mentions several design options in reference [16] that affect the communication between *Presentation* and *DialogKernel* (Figure 3). We based our description of the responsibilities on the architecture diagram of Figure 3; the displayed interfaces were considered in TABLE II accordingly.

3) *Synthesis and Description of the Graphical User Interface Software Category Model*

The resulting software category tree is depicted in Figure 10 and will be developed in the following paragraphs. It has to be considered that the software categories do model dependencies between units of design and no flow of events or algorithms. Although there will be interfaces between software categories for later implementation, these cannot be illustrated by the software category tree but can be later on determined concerning the possible type.

Principal units of GUI design. To clarify what units of design will be considered for a GUI system, we consulted the directions given by related work. Our findings were that MVC patterns often relate to single *Views* that model the visual display for a certain state of data or processing. In contrast, the Quasar client considers *Dialog* units that comprise of visual and logic components. Additionally, *Dialogs* feature an own life cycle and can activate or de-active each other, so that a flow of *Dialogs* and corresponding presentations or *Views* is established.

For a general GUI responsibilities model and its possible practical applications, the given definitions of both MVC and Quasar client were not entirely sufficient. As far as the Quasar client [16] is concerned, the relationship between input masks (or views) and dialogs is not entirely clarified, so that we received the impression that each Quasar client *Dialog* (Figure 3) is expected to have only one dialog data model and one *Presentation* (Figure 3) unit. As a consequence, we incorporated the following enhancements in the hierarchy of GUI design units:

A *Dialog* corresponds to one or more *Use Cases* of the system requirements specification and may be associated with several *follow-up dialogs* or *auxiliary dialogs* [16]. To provide data for display, processing and storing user inputs, each *Dialog* unit contains a *Dialog Data Model*. This model is closely related to the data requirements of the realized *Use Cases*. To be able to present several *Use Cases* steps individually or partition data among several views, each *Dialog* is associated with one to many *Presentation* units, which realize the corresponding display of a given *Dialog* or *Dialog Data Model* state.

From our experience, it is reasonable to keep dialog data and consecutive user interaction steps with several different displays together in one GUI design unit.

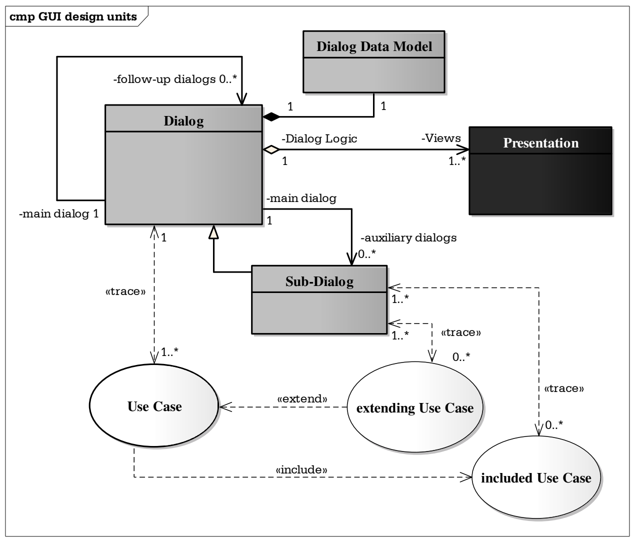


Figure 9. Principal units of GUI design and their requirement sources.

For instance, a *Dialog* may consist of a *Dialog Data Model* with several objects that cannot be displayed with one single window. Accordingly, the data is structured among several *Views*, which can be realized with different tabs of one window or with several windows. That is why each *Dialog* may reference several *Presentation* units, which serve as different *Views* (Figure 1) with their sets of UI-Controls and layout definitions. Accordingly, the user may proceed with required *Use Case* interaction steps straight forward or may return to previous steps in order to revise inputs. The data for all steps is kept together in one *Dialog* unit and its respective *Dialog Data Model*. Therefore, the communication needs between *Dialogs* concerning data exchange is reduced to a required minimum.

Furthermore, a *Dialog* may reference *Sub-Dialogs* that are closely related to either *included* or *extending Use Cases*. For instance, a search for certain objects can be added to some *Dialogs* as a *Sub-Dialog* to support the user during the selection of relevant data (*included Use Case*) for a certain context (*Use Case*). The particular search *Sub-Dialog* may appear in other *Dialogs* as well.

Figure 9 illustrates the GUI design units and their described relationships. The GUI design units were identified in correspondence to the event processing chain of Figure 2 and the basic software categories and layers of Section III.C.3) we apply for software category modeling. Thus, the *Dialog* serves the presentation logic and *Presentation* is responsible for presentation technology. Both GUI design units will lead the identification of detailed software categories and respective responsibilities within their scope of data and event handling.

The sub-trees of software categories illustrated by Figure 10 will be described with respect to their different scope as follows.

Presentation layout. The categories derived from *Presentation* are closely related to the *View* and *Controller* of the MVC pattern [18] and detail both their responsibilities. TABLE III provides a summary of the software categories modeling *Presentation* layout concerns.

Presentation is marked with FUI (final user interface) [43] 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*, *View Navigation* and *Presentation Event Handling*. The involved software categories have to comply with project specific dialog specifications and at the same time need to possess knowledge about the types and operations the integrated *GUI Framework* offers. Hence, all sub-categories heavily depend on technical aspects. They each constitute a mixed category.

TABLE III. PRESENTATION SUB-TREE SOFTWARE CATEGORIES.

Sub-Category	Topic	Contained entities	Operations
Presentation	Visual parts of a *Dialog* that realize the presentation technology layer. defines interfaces used in child software categories for construction purposes	Defined: Presentation Construction interface, View Definition interface, Presentation Event Handling interface	Abstract
Presentation Construction	constructor of a *Presentation* unit	Referenced: Presentation Construction interface, View Definition interface, Presentation Event Handling interface	Creation: *Presentation* (*View*) units with their comprising parts of View Definition, Presentation Event Handling and View Navigation Implementation: Presentation Construction interface (activates the constructor of a *View* to enable its creation along with associated UI-Controls, layout and event handling)
View Navigation	Changes and activates the *Views* that can be part of of one *Dialog* unit. This responsibility is essential for *Dialogs* that constitute several steps with or without different choices leading to certain *Views*. *Views* shall be decoupled from each other to facilitate their exchangeability and even reuse. That is why the *View Navigation* interface has to be called from outside the *Presentation*.	Defined: View Navigation interface, states or target *Views* for navigation Referenced: Presentation interface	Creation: Creates different *Views* by calling Presentation interface Implementation: View Navigation interface (offers access for entities outside *Presentation* to trigger *View* changes)
View Definition	Visual part of a *View* that creates and holds all UI-Control and layout information	Defined: UI-Control Configuration interface, Layout Definition interface Referenced: View Definition interface	Creation: UI-Control Configuration, Layout Definition Implementation: View Definition interface (constructor)
UI-Control Configuration	construction of UI-Controls, setting of UI-Control specific properties	Defined: possibly specialized UI-Controls created by inheritance from the GUI framework, UI-Control state data Referenced: UI-Controls of the GUI framework and their properties, UI-Control Configuration interface	Creation: create UI-Control Delegation: set UI-Control property Implementation: UI-Control Configuration interface (creates the UI-Controls upon being called by *View Definition*)
Data Display	UI-Control specific display of data, interpretation of model data [15] like coloring and highlighting of validation errors The dependency to the GUI framework *Technical Data Models* is derived from *Model Data Observer* and its parent software category *Presentation Data Handling*.	Defined: UI-Control display data (for simple data display and interpretation of data) Referenced: Dialog Data Model read interface Technical data model interface	Delegation: Read Dialog Data Model Algorithm: Interpret Dialog Data Model Control: change the technical data model and associated display of UI-Controls based on Dialog Data Model contents and its interpretation
Layout Definition	Creates and defines the layout of the *View* The category itself is abstract, so that its child software categories do the actual implementation of layout creation. Thus, the child categories can be regarded as different strategies of the *Layout Definition* interface.	Referenced: Layout Definition interface, Layout managers of the GUI framework, UI-Controls of the *View*	abstract
Arrangement of UI-Controls	Creates the general *View* layout, assigns layout to containers like panels, parts, cells positions UI-Controls inside layout containers	Referenced: Layout Definition interface	Implementation: Layout Definition interface (creates the *View* layout upon being called by *View Definition*) Algorithm: create *View* layout with the help of layout manager operations

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*.

View Navigation enables the change of different *Presentations* of a *Dialog* with respect to Figure 9.

Data Display was added to better reflect the visual presentation of data, which was formerly [1] included in *UI-Control Configuration* (setting properties for data values), and includes the interpretation of certain data values as an additional responsibility derived from [15].

Presentation event handling. The *Presentation Event Handling* category serves the task to receive and evaluate *Presentation events* according to Figure 2. It 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 how to react to any received events is concentrated in *Presentation Event Handling*. Its child software categories serve the above described responsibilities on demand of the superior evaluation of *Presentation Event Handling*. For instance, the decision about what respective events are to be forwarded is made by *Presentation Event Handling* and the actual forwarding command is encapsulated by *Event Forwarding*.

In TABLE IV, the software categories responsible for *Presentation* based event handling are summarized.

TABLE IV. PRESENTATION EVENT HANDLING SOFTWARE CATEGORY SUB-TREE.

Sub-Category	Topic	Contained entities	Operations
Presentation Event Handling	Event handler called by an UI-Control with active *Action Binding* This software category evaluates any incoming events from UI-Controls and decides on a proper reaction: *Presentation Data Handling*, *View State Changes* or *Event Forwarding* are triggered. For instance, it decides what events can and cannot be processed by the *Presentation* and must be forwarded to the *Dialog Event Handling*. Just the decision is covered here, how the forwarding is performed is in the scope of the respective child software category.	Defined: Event Forwarding Interface, View State Change Interface, Presentation Data Handling interface, Action Binding interface Referenced: Presentation Event Handling interface	Implementation: Presentation Event Handling interface (constructor), Action Binding interface (to be notified of any event intercepted by UI-Control Action Binding) Algorithm: Determine the proper reaction in response of the received event Control: Activate the proper reaction implemented by its child software categories
Action Binding	definition of various event listeners for UI-Controls to enable a reaction to specific events The event is just intercepted by the implementation of the event listener interface. Eventually, the resulting reaction is not covered but prepared with the delegation to the presentation event handling.	Referenced: Event listener interfaces of the GUI framework, Action Binding interface	Implementation: specific event listener interfaces of the GUI framework Delegation: call Action Binding interface to notify Presentation Event Handling about user inputs
Event Forwarding	forwards events to the *Dialog Event Handling*	Referenced: Event Forwarding interface, Dialog Event Handling interface	Implementation: Event Forwarding interface Delegation: forward event (Dialog Event Handling interface)
View State Changes	Interface that permits the change of *Presentation* states, which can be called by the *Presentation State Update*. May be called for changes like the activation of hidden or collapsed panels. The possible states a *View* can adopt are modeled by this software category.	Defined: Interfaces of child software categories (Re-Arrangement of UI-Controls, Modification of UI-Control Properties, Addition and Removal of UI-Controls), *View* state model Referenced: View State Change Interface	Implementation: View State Change Interface Control: Call appropriate child interface to enable the appropriate change of visual state
Re-Arrangement of UI-Controls	Change the position of UI-Controls inside the *View* layout on request of *View State Changes*	Referenced: Re-Arrangement of UI-Controls interface	Implementation: Re-Arrangement of UI-Controls interface Algorithm: alter *View* layout with the help of layout manager operations

Sub-Category	Topic	Contained entities	Operations
Modification of UI-Control Properties	activate, hide, or change UI-Controls in size, color or any other visual property on request of *View State Changes* May be called when data validation failed or new data state requires the update of particular UI-Controls only. In addition, UI-Controls can be set to be read-only when no further editing shall be permitted.	Referenced: Modification of UI-Control Properties interface, UI-Control state data, UI-Controls of the GUI framework and their properties	Implementation: Modification of UI-Control Properties interface Delegation: set UI-Control property
Addition and Removal of UI-Controls	change the set of active UI-Controls of a particular *View* on request of *View State Changes* UI-Controls may be added or removed as a result depending on the current data state or events evaluation.	Defined: possibly specialized UI-Controls created by inheritance from the GUI framework, UI-Control state data Referenced: Addition and Removal of UI-Controls interface, UI-Controls of the GUI framework and their properties	Implementation: Addition and Removal of UI-Controls interface Creation: create UI-Control, delete UI-Control Delegation: set UI-Control property
Presentation Data Handling	event handling that is only concerned about data changes and storage from the *Presentation* point of view	Defined: Model Data Edit interface, Model Data Observer interface Referenced: Domain Data Model, Technical Data Models of the GUI framework, Dialog Data Model interface, Dialog Data Model observer registration interface	Delegation: register as observer with *Dialog Data Model* Algorithm: determine proper data handling reaction Control: initiate data update via Model Data Observer interface, Activate Model Data Edit interface
Model Data Edit	changes *Dialog Data Model* in order to store user inputs present in active UI-Controls	Referenced: Dialog Data Model write interface, Model Data Edit interface	Implementation: Model Data Edit interface Delegation: change dialog data
Model Data Observer	retrieves data from the *Dialog Data Model* after being notified as observer of that model, loads data for *Presentation*	Referenced: Dialog Data Model read interface, Model Data Observer interface	Implementation: Model Data Observer interface Delegation: read dialog data

With respect to *View State Changes*, the Quasar client reference architecture [16] seems to miss an interface provided by *Presentation* that can be called by the *DialogKernel* to trigger changes like the activation of a dedicated panel that displays properties when the user performs a certain selection. Reference [14] states that this problem can be solved via an additional observer pattern instance.

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.

The applied branching of the *GUI Framework* serves the fine-grained presentation of dependencies, so that these model what detailed relationships the other software categories have with *T* software categories.

Dialog Logic. The last main category that is to be placed in the vicinity of a *Dialog* is the *Dialog Logic*. Software 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 [2][16] and the *Model* part of the MVC pattern [18]. 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 latter are based on *Dialog Data Model*, since dialog state evaluations largely depend on current *Dialog Data Model* states, which already reflect the inputs and choices the user may has actuated.

Dialog Data Model. The software 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 presentation logic layer (see Section III.C.3)).

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.

TABLE V summarizes the responsibilities that are concerned with handling *Dialog* data.

TABLE V. DIALOG DATA MODEL SOFTWARE CATEGORIES SUB-TREE.

Sub-Category	Topic	Contained entities	Operations
Dialog Data Model	establishes the data model used in the entire *Dialog* unit, serves as a global *Model* element according to MVC terms	Defined: Dialog Data read interface, Dialog Data write interface, aggregates or selections of business objects and their attributes (intermediaries [31]), additional data evaluation or interpretation information not present in *Domain Data Model*, list of observers Referenced: data read and write interfaces of *Domain Data Model* and data types (*Data Types and Validation Rules*), Presentation Data Handling interface (observer update), Dialog Data Model interface	Delegation: data read and write operations on the *Domain Data Model* and its data types, notifies *Presentation Data Handling* about data changes (observer pattern) Algorithm: offer browsing and selections of contained *Dialog Data Model* elements specific for display choice options Implementation: Dialog Data read interface, Dialog Data write interface, Dialog Data Model interface
Data Validation	validates *Dialog* data This responsibility may cover the comprehensive validation of multiple attributes or objects at once. Otherwise just the validation interface of individual objects is called and evaluated in order to provide validation information for the *Presentation*.	Defined: validation information beyond the scope of single business objects or data types Referenced: validation interface of *Domain Data Model* or its data types (Data Types and Validation Rules), Dialog Data write interface	Creation: validation information Algorithms: validation of *Dialog* data objects beyond the scope of single objects Delegation: call the validation interface of *Domain Data Model* or its data types Control: Change *Dialog Data Model* based on validation results
Data Conversion	offers transformations between technical and domain data model formats The *Dialog Data Model* may define new getters and setters that accept GUI *Technical Data Models* types or may trigger the call of a dedicated component (R software) [6] providing generic conversions.	Referenced: data read interface of *Domain Data Model* and data types (*Data Types and Validation Rules*) (derived from parent category *Dialog Data Model*), *Technical Data Models*	Algorithm: data conversion operations Delegation: data read operations from both data model formats

The *Dialog Data Model* serves as the primary *Model* according to MVC terms; UI-Controls do only hold their properties that mirror small parts of the *Dialog Data Model*. Furthermore, observer functions are considered *0* software and can be included anywhere, so they require no special interfaces. For the sake of completeness, selected operations have been included in TABLE IV and TABLE V.

Dialog 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: presentation logic tends to be stable (enter data, evaluate, present suggestions, make a choice and confirm), is traced to functional requirements (see Figure 9), and thus, should be decoupled from GUI layout specifications. Although the flow of presentation logic is unaffected, the GUI and its technology supporting the user in his tasks may be altered several times starting with updated visual 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 (or *Dialog Data Model* of Figure 9) and technological or concrete presentation states, which implement the former by using visual appearances. 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 presentation 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.

The responsibilities for dialog event handling and respective software categories are summarized in TABLE VI.

TABLE VI. SOFTWARE CATEGORIES RESPONSIBLE FOR DIALOG EVENT HANDLING.

Sub-Category	Topic	Contained entities	Operations
Dialog Logic	The software category and its children are responsible for the presentation logic part of a *Dialog* that connects application logic and presentation technology. Defines interfaces used in child software categories for construction purposes.	Defined: Dialog Data Model interface, Dialog Event Handling interface, Dialog Logic Construction interface	Abstract
Dialog Logic Construction	constructor of a *Dialog Logic* unit	Referenced: Dialog Data Model interface, Dialog Event Handling interface, Presentation Construction interface, Dialog Logic Construction interface	Creation: *Dialog* units with their comprising parts of *Dialog Data Model*, *Dialog Event Handling*, *Presentation* (initial state of a *Dialog* is created) Implementation: Dialog Logic Construction interface
Dialog Event Handling	definition of *Dialog* states and associated actions It is computed what actions are allowed (reload data, confirm) in a given *Dialog* state and how the *Dialog* is altered because of received events. The results or reactions of the *Dialog Event Handling* are each modeled by child software categories: *Dialog Lifecycle Actions*, *Application Server Calls* or *Presentation State Updates* are activated, which enable different behavior or control states of other lower situated entities (*sub-dialogs*, *follow-up dialogs*, *Presentation*). However, the parent category *Dialog Event Handling* resumes the task to decide what child category is finally called in a certain *Dialog* state. In some *Dialogs* data evaluations are needed to trigger the proper *View* from several configurations, which may be rule-based. In this regard, the logic required for changing pages in large scale *Dialogs* like wizards when data was validated successfully is modeled by this software category. The evaluation is done by the *Dialog Event Handling*, but the actual change of *View* is performed by *Presentation State Update*. The latter receives the command to just switch to a certain *View*. The decision to what view is to be switched lies in the scope of *Dialog Event Handling*. Please note that the branching of *Views* is not assigned to the *Dialog Data Model*, since the model can be reused elsewhere with different rules for navigation or display.	Defined: dialog state model, dialog event forwarding interface, dialog event reaction interfaces (Dialog Lifecycle Actions interface, Application Server Calls interface, Data Queries interface, Presentation State Update interface) Referenced: Dialog Event Handling interface, Dialog Data Model	Creation: dialog state model Algorithm: evaluate current *Dialog* state and determine appropriate reactions (e.g., evaluate *Dialog* state on the basis of *Dialog* data in order to determine navigation options) Implementation: Dialog Event Handling interface (constructor), dialog event forwarding interface (called by *Presentation Event Handling* to notify about events to be processed) Control: Call appropriate event reaction interfaces, proper sequences of *Application Server calls* or *Dialog Navigation*
Dialog Lifecycle Actions	construction of *Dialog* units, changes global states of current and other *Dialogs* The scope of this category is the reaction on special events like OK, Cancel and similar terminal notifications. As a result, an entire *Dialog* unit is created or discarded. The associated design units represented by *Dialog Data Model* and *Presentation* are created indirectly by activating a cascade controlled by the *Dialog Logic* and its states. In addition, other *Dialog* units may be ordered to be activated or de-activated by calling the *Dialog Navigation* interface.	Defined: Dialog Navigation interface Referenced: Dialog Logic, Dialog Lifecycle Actions interface, Dialog Logic Construction interface	Creation: Dialog Logic creation / deletion (*Dialog Data Model* and associated *Presentation* are created or deleted implicitly) Implementation: Dialog Lifecycle Actions interface (called by *Dialog Event Handling*) Control: determines the proper sequence of *Dialog* units to be activated and de-activated (Dialog Navigation interface)
Dialog Navigation	performs the navigation among *Dialogs* or activation of *Sub-Dialogs* The opening and closing of auxiliary *Dialogs* like search dialogs for master data (e.g., customer ID and address) is performed.	Referenced: Dialog Navigation interface, Dialog Data Model, Dialog Logic Construction interface (other *Dialog* instance units)	Create: Create and discard *sub-* or *follow-up dialogs* Implementation: Dialog Navigation interface
Dialog State Changes	addresses the possible changes in state with respect to the currently active *Dialog* only	Abstract	Abstract

Sub-Category	Topic	Contained entities	Operations
Application Server Calls	event handling routines that interact with the *Application Logic* services This software category models the reactions on particular events that require the activation of services of the *Application Logic*.	Referenced: Application Server Calls interface, Application Services interface, Domain Data Model	Implementation: Application Server Calls interface Delegation: Application Services interface
Data Queries	loading and updating domain layer data As a specialization of *Application Server Calls*, the retrieval and sending of data in correspondence with the interfaces of *Application Services* is of particular interest.	Referenced: Data Queries interface, Application Services interface, Domain Data Model, Dialog data interface	Algorithm: Assembly or selection of appropriate data queries provided by *Application Services* Implementation: Data Queries interface Delegation: Proper calling sequence of *Application Services* for data retrieval Control: Setting *Dialog* data
Presentation State Update	triggers the change of *Presentation* states / visual layout	Referenced: Presentation State Update interface, View State Change interface, View Navigation interface	Implementation: Presentation State Update interface Delegation: calling of state change notifications of the *Presentation* (View State Change Interface, View Navigation interface)

The interfaces that connect the software categories for event handling are to be defined in detail as reusable *0* or *A* software (much like the observer pattern [17]). That is why there are no dependencies visible in Figure 10 between *Dialog Event Handling* and *Presentation Event Handling*. The same applies for the visibility between *Presentation State Update* and *View State Changes* or *View Navigation*. Finally, a command [17] interface may be used that contains only stereotype operations and can be typed as *0* software. Each of the involved event handling software categories is implicitly connected to *0* software via the various parent software categories in the hierarchy.

Please note that the parent software categories of *Dialog Event Handling* and *Presentation Event Handling* define most interfaces for their children, so that they are able to control them but do not depend on their detailed actions, internal types or implementations. The children encapsulate the results of a response chosen by the parent category for a certain event.

From the presentation logic's perspective, a *Dialog* may adopt different states during runtime. The required knowledge to enact these states is represented by the abstract category *Dialog State Changes*: only its refinements will be assigned to design units; the parent software category *Dialog State Changes* serves grouping purposes and summarizes commonalities of the children. *Dialog State Changes* is separated into children, 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 Event Handling* possesses the knowledge how to react in a given situation. Its children are dedicated to solely apply the required change of state that either addresses the *Application Server* or *Presentation*, which provide the state change execution. Thus, the children and other server-like entities (e.g., *Application Services*, *View Navigation* and *State Changes*) do not know when their services are called.

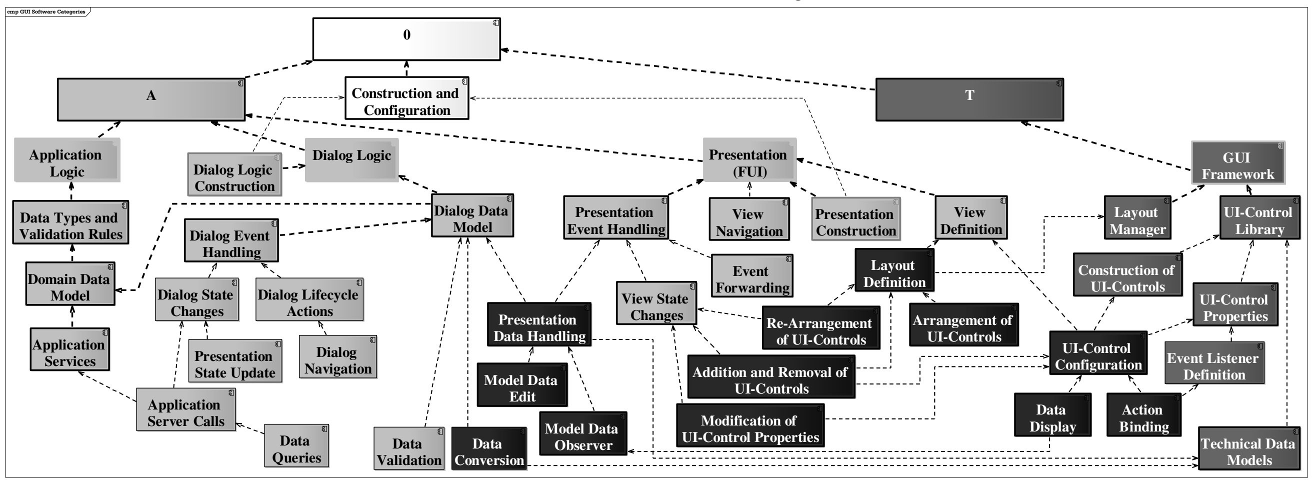


Figure 10. GUI responsibilities modeled as a software category tree.

4) Object Lifecycles and Construction

In this Section, we briefly describe how the construction of instances is considered by the software categories of Figure 10.

As we learned from Figure 9, there are the principal GUI design units *Dialog*, *Sub-Dialog* and associated *Presentations*, which will bear the major part of responsibilities in real GUI systems. To lead to creation of these units, we have incorporated constructor responsibilities within the software category tree that compare to the *DialogManger* of the Quasar client (see Figure 3). Particularly, the *Dialog* and *Presentation* both were supplemented with responsibilities dedicated to construct the child elements of these parent software categories.

In this regard, the *Dialog Logic Construction* is responsible for the creation of the main *Dialog* unit. We assume that a *Dialog* design unit will correspond to the software category sub-tree modeled by *Dialog Logic*. Based on the responsibilities a *Dialog* has to fulfill, it initiates the construction of the starting *Presentation* as an entry point for user interaction after the creation of own member objects. This sequence is to be followed, since the *Dialog Logic* controls the states of the *Presentation* anyway.

Concerning the *Presentation*, this design unit also features a software category (*Presentation Construction*) dedicated to the construction of its child elements.

Both the *Presentation* and *Dialog Logic* may call the construction of additional units of their type when respective events occur: for the *Presentation*, new *Views* will be requested by *View Navigation* upon a call from *Presentation State Update* was received. With respect to the *Dialog Logic*, during the event processing by *Dialog Event Handling* a *Dialog* may be finalized or a new *Dialog* instance may be created as a result of a *Dialog Navigation* event reaction. Both options are controlled by *Dialog Lifecycle Actions*.

Figure 11 provides an overview about the dependencies concerning lifecycles and construction of instances based on the software categories of Figure 10.

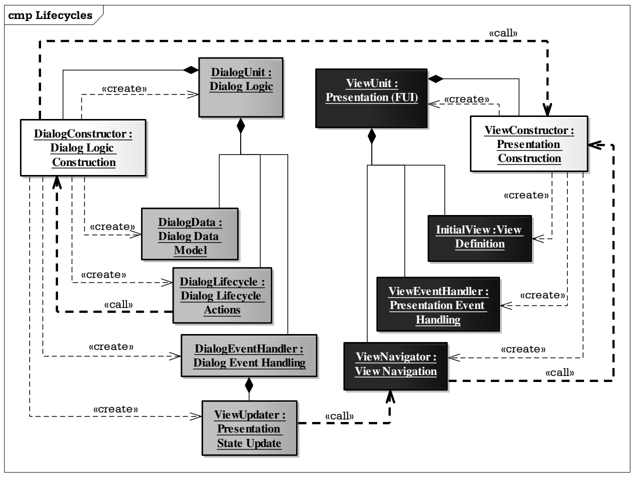

cmp Lifecycles
«call»
DialogUnit : Dialog Logic
«create»
ViewUnit : Presentation (FUI)
«create»
DialogConstructor : Dialog Logic Construction
ViewConstructor : Presentation Construction
«create»
«create»
DialogData : Dialog Data Model
InitialView :View Definition
«create»
DialogLifecycle : Dialog Lifecycle Actions
«call»
ViewEventHandler : Presentation Event Handling
«create»
«create»
DialogEventHandler : Dialog Event Handling
ViewNavigator : View Navigation
«create»
«call»
«create»
ViewUpdater : Presentation State Update
«call»

Figure 11. Intended lifecycle dependencies and constructors of possible objects derived from the GUI software category model.

Whenever new instances are to be created, an object that implements the respective construction responsibility of either *Dialog Logic Construction* or *Presentation Construction* is to be delegated.

5) The Event Processing of the Software Categories

Figure 12 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. Basically, three different scopes for states are modeled by the software categories. They are the following:

- *Dialog Logic - Application Services*: The scope of this state model is concerned with the data model of the entire *Dialog* unit and the interaction with *Application Services*. Decisions are to be taken what services and data contents are to be combined for the required interaction of a given *Use Case*. As a result of the *Dialog Logic* state model evaluation, a change of the visual state may need to be delegated. It depends on the GUI specification with respect to the required steps a given *Use Case* scenario may demand for.
- *Dialog Logic - Presentation*, *View* level: A *Dialog* may require consecutive *Views* to be displayed in a certain sequence or based on user decisions. These changes of *Views* are in the scope of a dedicated state model.
- *Presentation*, UI-control level: The different states a particular *View* may adopt are considered herein. This covers different changes in layouts and UI-Control configurations.

The general flow of events is indented to work as follows: initially, the user triggers some events that are intercepted by UI-Controls that have an *Action Binding* configuration. In any case, the event is passed on via *PresentationEventHandlingInterface* to the *Presentation Event Handling*. A first evaluation of that event may result in a decision by *Presentation Event Handling* to further move the event on the event processing chain via *EventForwardingInterface* to *Dialog Event Handling* for the final evaluation.

Depending on the current state of the *Dialog*, *Dialog Lifecycle Actions* (creation and deletion of *Dialogs* and their objects), a *Dialog Navigation* (change of current *View* or the instantiation of *Sub-Dialogs*), *Application Server Calls* (commit a sequence of service calls) or a *Presentation State Update* (change of the visual representation) may be activated as reactions by *Dialog Event Handling*.

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*. Please remember that *Presentations* or *Views* may be reused in different contexts (compare pluggable *Views* in reference [31]), and so, a direct binding of their UI-Control events to state changes would greatly limit their flexibility and adaptability. Therefore, the event firstly is forwarded via the *DialogEventHandlingInterface* interface of Figure 12. Then, the *Dialog Event Handling* evaluates the event and controls one of its children, which further delegates to the displayed interfaces of Figure 12 and initiates the final change of state. Concerning the *Presentation State Update* in Figure 12, *View State Changes* (panels are activated) or *View Navigation*s (wizard steps or tabs are switched) are committed via interfaces. Another option would be a change of the *Dialog's* lifecycle or even a *Dialog Navigation* (separate *Dialogs* or an auxiliary search *Dialog* are instantiated) could be performed.

In this context, the knowledge when to trigger any of the interface operations is kept in the parent category *Dialog Event Handling*. In contrast, the execution of the respective state change is encapsulated in the child categories, which are marked by a white border in Figure 12 and implement the interfaces. At last, the state changes are completely decoupled from the point in time when they are requested.

Moreover, 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. Furthermore, 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.

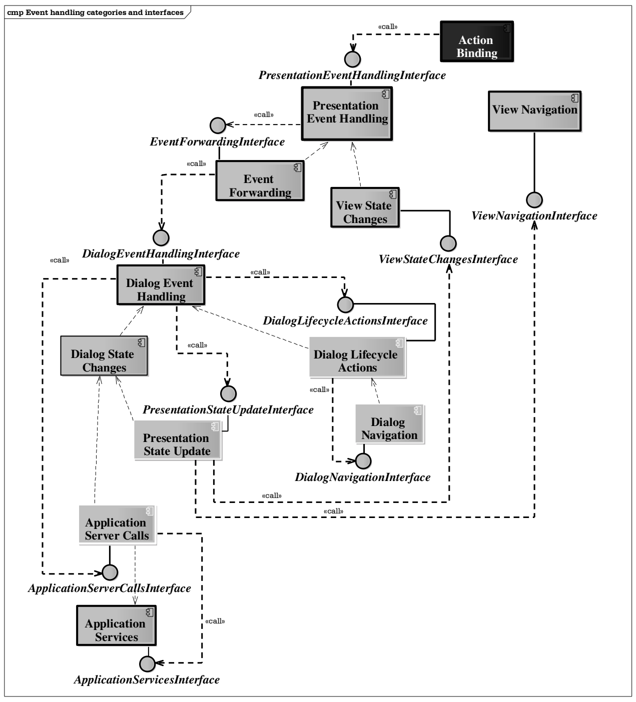

cmp Event handling categories and interfaces
«call»
Action Binding
PresentationEventHandlingInterface
Presentation Event Handling
View Navigation
«call»
EventForwardingInterface
«call»
Event Forwarding
View State Changes
ViewNavigationInterface
DialogEventHandlingInterface
ViewStateChangesInterface
«call»
«call»
Dialog Event Handling
DialogLifecycleActionsInterface
Dialog State Changes
«call»
Dialog Lifecycle Actions
«call»
PresentationStateUpdateInterface
Dialog Navigation
Presentation State Update
DialogNavigationInterface
«call»
«call»
Application Server Calls
ApplicationServerCallsInterface
Application Services
«call»
ApplicationServicesInterface

Figure 12. Software categories relevant for event processing and possible interfaces.

To conclude, the event handling approach and its respective software categories ensure that the layers of presentation technology and logic (introduced in Section III.C.3)) remain strictly separated. In fact, there will be dependencies among *Dialog Logic* and *Presentation* that cannot be avoided like the consistency of logic and visual states. However, the control of all states remains centered in one unit of design (*Dialog Logic*), which will facilitate development and maintenance of complex *Dialogs*.

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 work published in [4] and [10], 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.

1) 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*.

2) Issues

According to the MVC variants, we see major design issues that will be described in the following paragraphs.

Separation of concerns. To begin with, 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 [31]. Therefore, the component with concentrated tasks tends to be overburdened, and finally, can end up as the bottleneck from a maintenance perspective. Additionally, altering the tasks of the three components in certain variants may result in a simplification of one component that can only be employed for stereotype tasks but fails to suit more complex scenarios. There seems to be no ideal separation of concerns among the three components. A fourth element may be missing.

In general, there are no hints given how the display for certain portions of business logic or data can be decoupled from their technical manifestation. More precisely, the *View* part is directly coupled to the *GUI Framework* (Figure 1). 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 and related control. With regard to the event processing chain of Section II.B, the GUI patterns do not distinguish clearly between events related to technical or application concerns. In general, a guideline is missing for the decision when to shift between presentation technology or presentation logic related processing of events. TABLE I provides an overview about the assignment of these layer specific responsibilities to MVC pattern roles.

Although the MVP variants [6][7] and HMVC [5] employ a "Supervising Controller" [15], which receives each event from any UI-Control and acts as a global MVC *Controller*, the problem persists: the *Presenter* as well as the HMVC *Controller* still have to decide whether the incoming events require an presentation technology or presentation logic specific processing and have to react accordingly. Yet, these approaches solve the "visibility problem" described by Karagkasidis [10] where the *Controller* and *View* are separate classes. In any case, the developer has to refine the architecture by himself to enable a differentiated handling of presentation and application related events. Finally, the reuse may be affected, since the *Controllers* end up processing both types of events for the sake of initially quick releases.

Cohesion and granularity of triads. With the application of MVC derivates that differ from the classic MVC approach [31] a problem occurs 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 proper modularization of *Dialog* components is to be done on behalf of the developer. Only the HMVC [5] gives some rudimentary hints. In the end, the general size and scope of MVC triads is not clear. According to Karagkasidis [10], a *View* may constitute of single UI-Controls (widgets), containers like panels with a certain set of UI-Controls or complete *Dialogs*. The classic MVC approach [31] was clearer on that topic, since MVC triads were very fine-granular starting at the UI-Control (widget) level and building a corresponding triad for every element of the visual object hierarchy, ultimately ending with a last triad at the window level. However, the classic approach is not likely to be feasible for modern and more complex application scenarios: the high integration of business systems and their complexity would demand for a large number of *Dialogs* that would result in myriad of MVC triads.

Coupling of *Controllers* to both *Model* and *View*. With respect to the above described limited separation of concerns more issues arise. The controlling of both *Presentation* states and the handling of application related events to initiate *ApplicationKernelService* calls inside the *Controller* creates close coupling of *Controllers* to both *View* elements and naturally the *Model*. 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 (rather than representing separated classes) 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 entire *Dialog* alter as well, so that the *Controller* implementation may have to be revised each time. This is due to the awareness of *Controllers* about the *View's* UI-Controls what results from the following. In modern GUI frameworks the *Controllers* obtain user entered data directly from UI-Controls and not as the payload of an incoming event, as this was the case in Smalltalk or classic MVC [31]. With the latter, separate classes for *View* and *Controller* could be realized but current GUI frameworks demand for alternative solutions. Karagkasidis [10] exemplarily discusses the solution provided by HMVC.

To partly resolve this issue and decouple the *Controller* at least from application aspects, a developer could revert to the "Model as a Services Façade" [4] 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 [44] or use the domain model pattern [32] 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* [4], 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 selections, validation state) 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 a *View*, its display, as well as in- and outputs.

3) *Summary*

The MVC and its derivates require much adaptation in order to be prepared for implementation [14]. The above mentioned issues may considerably have a negative impact on the resulting architecture quality. The available patterns are definitely not easy to interpret with respect to the common set of GUI responsibilities illustrated by the software category tree in Figure 10.

The tracing of functional requirements to the parts of the GUI, which coordinates *ApplicationKernel* service calls, will largely depend on the refinements the developers have incorporated in the GUI architecture. Additionally, a clear separation of presentation technology and logic (see Section III.C.3)) is not supported in any variant, so that event handling will always consume high efforts for development and especially maintenance.

Anyway, 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 a challenging goal hard to achieve with available patterns. Project budgets may severely limit the software architecture quality to be attained.

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.

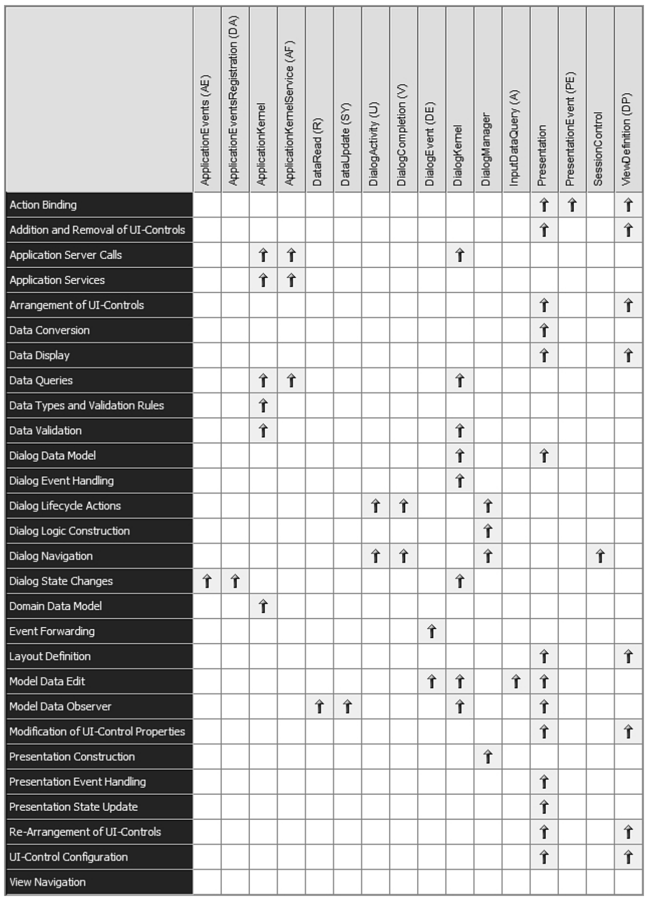
Positive aspects. In contrast to the MVC variants, the Quasar client separates *Presentation* and *DialogKernel* as principal dialog components. This separation is the main source for its virtues, since more clearly distinguished *Controller* tasks are achieved. In this regard, the *Presentation* is 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 [16], 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 [17] pattern [14]. 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 received from the *DialogKernel* via *DataUpdate* interface. The triggering of visual state changes on behalf of the *DialogKernel* (*Presentation State Update*) may be possible that way 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. According to Siedersleben [16], the states of visual elements are exclusively controlled by the *Presentation*. However, in the particular example only the *DialogKernel* would possess the knowledge when to trigger the state change of the *Presentation*. It seems that the cooperation of both units of design needs further elaboration to be able to be implemented in practical examples. 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 13 displays the resulting traceability matrix. The sources for traceability-links constitute software categories of varying detail arranged on the left hand side.



	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 Display													↑			↑
Data Queries			↑	↑						↑						
Data Types and Validation Rules			↑													
Data Validation			↑							↑						
Dialog Data Model										↑			↑			
Dialog Event Handling										↑						
Dialog Lifecycle Actions							↑	↑			↑					
Dialog Logic Construction											↑					
Dialog Navigation							↑	↑			↑				↑	
Dialog State Changes	↑	↑								↑						
Domain Data Model			↑													
Event Forwarding									↑							
Layout Definition													↑			↑
Model Data Edit									↑	↑		↑	↑			
Model Data Observer					↑	↑				↑			↑			
Modification of UI-Control Properties													↑			↑
Presentation Construction											↑					
Presentation Event Handling													↑			
Presentation State Update													↑			
Re-Arrangement of UI-Controls													↑			↑
UI-Control Configuration													↑			↑
View Navigation																

Figure 13. The GUI software categories traced to Quasar client reference architecture components and interfaces.

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 (compare delegation and control of Section III.C.4)) are the more likely is the mixture of software 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 [2].

Separation of concerns. For the valuation of both cohesion and separation of concerns two directions inside the traceability matrix of Figure 13 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 software 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*, *Model Data Edit* and *Model Data Observer* are realized by several Quasar elements, and thus, different units of design.

The first three software categories mentioned before 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* may know the *DialogKernel* and its data interface (see association in Figure 3) but not vice versa [16]. As an alternative, newly entered data may be included as payload of the event emitted via *DialogEvent* by the *Presentation* [16]. How the important task of changing dialog data is performed in detail by the *Presentation* and what interfaces are required is finally left open.

Moreover, *Dialog Lifecycle Actions* are of less importance. They are rather stereotype operations that could be detailed by *0* type software. In contrast, 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 [45] would affect the software category assignments. Maybe the *Session* cannot be marked as *0* software anymore, since it would need knowledge of the proper sequence of dialogs, and thus, would be designated as *A* software that could not 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 data 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*, *Data Queries*, the *Dialog Data Model* 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 *Data 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 certain *Views*, so that they are less flexible for change and reuse. In addition, events can only be emitted by *View* elements and cannot 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.

A look at the matrix of Figure 13 reveals that the event handling of the Quasar client architecture with respect to presentation technology and logic concerns seems not to be elaborated with the necessary care and accuracy; there are several responsibilities mixed within and among *Presentation* and *DialogKernel*: firstly, the *Presentation* is in charge of both receiving events (*Presentation Event Handling*), deciding on visual states (*Presentation State Update*) and executing them (e.g., *Addition and Removal of UI-Controls*). Secondly, the needed knowledge for decisions, and thus, presentation logic is likely to be based within the *DialogKernel* as far as the interaction with the *Application Services* is concerned. Yet, the latter is assigned to handle its own state model (*Dialog Event Handling*) and partly manages the *Dialog* data (*Dialog Data Model*) together with *Presentation*. So, both design units share the information necessary for deciding upon state changes. In contrast to the GUI software category model of Figure 10, the Quasar client architecture assigns state decisions and executions based on a different point of view: presentation logic is strictly separated between application (*DialogKernel - Application Server Calls*) and visual behavior (*Presentation - Presentation State Updates*), so that the *Dialog Logic* and its state model is not centered but shared among two design units. For that reason, with the Quasar client a *Dialog* will be harder to adapt to a changed *Use Case* scenario affecting the *Dialog* state model (a new step with a new or updated *View* is required), since the *Presentation* is designed to both manage and execute the *View* state changes. So, the presentation logic required for deciding on a change or update of the *Views* is lost and has to be re-implemented whenever the *Presentation* has changed. From our point of view, a centralization of event-based decisions found in the GUI software category tree of Figure 10 would reduce the portions of *AT* software existent in any *Presentation* and could partly facilitate the exchange of *Views*.

As far as the visual part of the *Presentation* is concerned, 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* and its children.

Lastly, the *Presentation* is assigned quite a are large set of responsibilities, but is the design unit that is not likely to be stable or reusable after technological changes compared to the *DialogKernel*, which does not depend on any *T* software influences.

Missing responsibilities. Responsibilities that were entirely not mentioned with respect to the Quasar client reference architecture, was the *View Navigation*. This task may be confused with *Dialog Navigation*. Siedersleben approaches the architecture of a *Dialog* with the definition of the relevant terms in reference [16], but he does not use them in a consistent way, so that some terms are only mentioned and remain unrelated to the Quasar client architecture itself. As a consequence, the design unit of a *Dialog* remains unclear with respect to the delimitation of other *Dialog* instances, *Sub-Dialogs*, and more urgently, *Presentations* or *Views* of Figure 9 that express the different interaction steps with a user.

3) Summary

Our review of the Quasar client revealed that this reference architecture is more advanced than common GUI patterns. It includes most of the common MVC pattern responsibilities (TABLE I) and adds several additional ones (TABLE II). Besides, 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 [16][29][38][14] only available in German language. There was no comprehensive or updated description published, which would provide the needed implementation details. 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 of Figure 13 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 reference [2], 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 alone.

V. Results and Discussion

1) GUI Responsibilities Software Category Tree

One of our objectives was to provide a software category tree with separated concerns to describe a complete decomposition of GUI architecture responsibilities.

Software category model. We derived a software category model that structures the dependencies among common responsibilities of GUI architecture design units without being biased towards a certain GUI architectural pattern or framework.

Software category definition and modeling. To be able to model detailed, refined software categories and finally delimit them, we had to invent modeling rules that were not provided in the original sources. We are convinced that these enhanced rules create a solid foundation for modeling responsibilities of software architectures, since the results make sense in our case of a better understanding of GUI architecture patterns and bring us further towards UIP integration.

Compared to the CRC method applied for the GUI patterns in [18], the collaborators of a certain software category are summarized in the second dimension but are further outlined by the association with detailed operations. On the CRC cards, every responsibility of a design unit is noted on one card and there are not details about their relationships to the mentioned collaborators on that card.

Nevertheless, there are not only positive aspects about the software category modeling approach. In fact, there are some weaknesses of the software categories tree display: For instance, there is no hint what elements are actually derived from the dependencies of parent software categories. Generally, there can be all included or referenced entities or only a sub-set of them considered in the child software category. Some contained entities can even be derived from the parents of a parent category (e.g., *Data Display* - *Model Data Observer* - *Presentation Data Handling* - *Technical Data Models* is an example of such a cascade of dependencies or refinements to be discovered in Figure 10). Moreover, there is also no hint, which parent categories are skipped and will not be considered in child software categories. In most cases *0* software is used and almost never skipped, but along the way up to *0* not all software categories are always considered. Some relationships just model the potential visibility of entities. Maybe the detailed modeling of instance based software category trees can remedy some of these aspects by providing further detail.

Shape of the tree. 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. In this regard, is has to be noticed that the software categories displayed here are pure and intended to be well separated. This arrangement of responsibilities is mostly not the case in real systems and designs; the software category tree is an ideal construction.

Software architecture relationship. From our point of view, the different MVC pattern variants are hard to understand with all their facets concerning detailed responsibilities and dependencies on other design units they need to interact with. Often the MVC variants compose of smaller patterns like “Supervising Controller” [15] and “Presentation Model” [15], which are a proof of the ever present complexity of GUI design.

To partly address the complexity issue, the software category model presented in this work aims to display the responsibilities of GUI architectures without favoring certain structuring or role assignment of design units. They are created to provide an overview of the general responsibilities that may occur in GUI systems instead. Architects and developers shall get a guide what tasks are to be fulfilled within the GUI system.

There may be an inherent or obvious structure hidden in the separated sub-trees with *Presentation* and *Dialog Logic*. However, this structure simply emerges from the dependencies of knowledge (modeled by the dimensions of Figure 8), which is required for the different responsibilities. The displayed separation or decomposition of software categories has not to be strictly followed; there is rather high degree of flexibility: the software categories can be distributed differently to design units. For instance, the *Data Conversion* responsibility is often differently solved in designs. Some responsibilities may be omitted when requirements do not demand for them. Eventually, the resulting distribution of software categories to design units determines the final quality of the software architecture.

In this regard, architects can consult this model without the need to be restricted by given designs, their roles and relationships. The descriptions and sources used for the composition of the software category tree are not entirely distracting or misleading, yet they are quite helpful for understand certain designs. But their weakness is that they are already biased towards a certain structure of design or effects to achieve like this was elaborated by Alpaev in [4] for the MVC design options.

Software category refinement level. One may argue that the consideration and segregation of software categories may overburden an architect with additional tasks and he will eventually loose overview due to the management of a set of fine-grained responsibilities. In contrast, the software category tree shall be helpful and not a burden. In fact, the software categories build on the refinement from basic to detailed categories in a hierarchical tree. So, the architect principally can decide on the level of detail he applies for modeling, mapping or assessment of design. In this regard, software categories always group several responsibilities into a family of cohesive entities; children retain the more detailed responsibilities and parents serve as a more general aggregation. In that way, an architecturally meaningful re-composition of GUI responsibilities is created. The architect may pick a certain detail level of the category tree, which ideally resembles a prepared separation of concerns in any case, in order to re-distribute these responsibilities in a new system design. This choice decides whether only basic software categories are used for architecture planning or refined ones are applied instead in order to achieve a much better accuracy for cohesion as well as the evaluation of how well concerns were separated.

Software architecture assessments. Furthermore, the software category model can be of aid for the valuation of the detail, cohesion and separation of concerns of reference architectures or patterns. Section IV.B outlined the principal approach and an example that assessed the Quasar client reference architecture. In sum, the software categories approach can reveal not supported tasks, design units that bear many tasks at once, perfect matches and tasks that are shared among two or more units of design.

In our opinion, the established software category tree is well-suited for GUI architecture assessments: the software categories embody a set union of the responsibilities of many of the common GUI architecture patterns. In the context of GUI design, the software categories resemble different and delimited packages of knowledge, which are used to identify and map components or smaller units of design. Later on, the dependencies among the software categories will lead the design of interfaces between components [16] to achieve a minimum of coupling based on the rules established in reference [16]. Thus, the proper distribution of identified software categories among design units can have an enormous impact on software quality. During assessments, this intended way of identifying design units and delimiting them by assigning distinct tasks to them can be reversed. This enables an evaluation of the rationale the design is based on.

Available architecture patterns cannot provide a comparative view on GUI responsibilities, since they miss some details, are interpreted differently among developers, can be biased towards a certain programming language, and the discussion of their trade-offs is limited to their scope, so that the impact on the general architecture can only be partly valuated. In addition, patterns often need to be combined within a design, so that their different effects depend on the actual combination and their adaptations.

Interface design. 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 [16], an interface ideally should be defined on the basis of a software category that serves as a parent for both software categories to be linked. That way, the least coupling is ensured. Not always can a shared parent software category be found to serve as a basis for an interface between components. This may be due to an improper distribution of responsibilities among design units. As a result, the underlying software category model needs to be revised. Anyway, the identification of design units and their interface structure requires some detailed planning.

Relationship to implementations. The responsibilities modeled by the software category tree can be used to analyze and reflect implementations. According to Quasar references, this is only done on the level of the very basic software categories *0*, *A*, *T* and *AT*. With the now available refinements for GUI architectures, an actual design or implementation can be evaluated concerning the correspondence to software categories. Thus, the cohesion and separation of concerns ca be assessed. The other way around, given implementations may refine the software category tree and it could be practically examined if the visibility is sufficient moving the tree upwards starting from a certain category or if additional dependencies have to be modeled.

Missing concerns. Currently, concerns like user profiles, additional assistance, session management [14] 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.

Summary. By the application of software categories, the GUI responsibilities to be identified have been ordered and grouped according to their knowledge and purpose, but this was modeled independently from any specific software architecture. The software categories in that role are suitable to represent a set of GUI responsibilities without the need to mention specific data types or operations of certain frameworks. Finally, the way how frameworks are applied shall be adapted to the required set of responsibilities as well as the software architecture based thereupon and not vice versa.

2) *Major Issues in GUI Architecture Design*

Our first objective was to identify GUI design issues. These issues naturally result from points of improper coupling, non-separated concerns and in general missing responsibilities not modeled by available GUI architectures or patterns. We had to analyze the available architectural patterns, which 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 [46]. We analyzed the differences or missing details of presented architectural patterns and identified four major design issues that may have a considerable impact on GUI development and maintenance.

Presentation logic and application control flow. 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 *Dialog Logic* or *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 [16][14]. 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 [47]. The coordination of the various algorithms and delegations, 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 [16]. 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, service hierarchy), *DialogKernels* (logic for dialog flow and control of user interaction) and finally *Presentations* (visual part, in- and output UI-Controls, realization of visual states). 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.

Dialog navigation. This leads us to the second issue that is concerned with the flow of *Dialog* units or navigation among them. Karagkasidis [10] already described this issue from the perspective of an example with opening and closing *Sub-Dialogs*. Important aspects mentioned by Karagkasidis are the lifecycle management of *Sub-Dialogs* that can be related to our presented GUI design units of *Dialog, Sub-Dialogs* and *Views* from Figure 9: they need to be controlled by a dedicated entity that is able to assign data to them, which is appropriate in a certain context. In addition, events from every GUI design unit of the hierarchy, which are significant for the further event handling or application data, have to be integrated in the presentation logic flow or event processing chain, so that individual units do not act isolated but create a comprehensive sequence of events.

Recent research [48][49] 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*.

Large AT software portions. 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 regularly occur. The *Presentation* communicates with both the *GUI Framework* and *DialogKernel* in order to retrieve and store 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 be invented. An initial approach was formulated by Siedersleben and Denert in [29].

Granularity of GUI pattern design units. Another GUI design issue could be identified that is cross-cutting along the previously described three GUI design problems. It is concerned with the proper sizing of GUI design units, or with respect to common GUI patterns, MVC triads [10]. In detail, the main objective is keeping the event processing chain of the GUI perfectly matched with the functional requirement side of the value creation chain represented by business processes and corresponding use cases. Ultimately, these two mental models of event flows have to be kept in close synchronization to be able to firstly realize requirements properly and secondly apply changes to the GUI system efficiently when requirements are altered or added. Simple MVC or even greater HMVC [5] or MVP [6][7] *Controllers* are quickly overburdened in their scope in the attempt to trace functional requirements of the value creation chain, and so, keep track individual steps of application control flow.

The introduced GUI software categories (Figure 10) shed light on the granularity problem as they clearly distinguish greater and lesser components like *Dialog Logic*, *Presentation*, *View Definition* and *Presentation Event Handling*.

Originally, the MVC and its derivates were not designed to address such complex and hierarchical structures within information systems. Please remember that the classic MVC was built with the assumption in mind having this architecture applied as the global architectural style: there were no additional units of application or domain related design (generally *A* software descendants in terms of Quasar software categories) besides *Models*.

Nowadays, application and presentation logic as well as business processes do pose a difference to that rather simple *Model* design of the past. Therefore, *Controllers* face a different scope inside the value creation chain. To be able to separate concerns and keep a high cohesion, *Controllers* need to be assigned a proper level of responsibilities within the GUI software category tree. This in turn requires a corresponding sizing of triads or other pattern based GUI design units.

Identification of GUI design unit instances. Besides the granularity problem, there is an additional conflict whether to provide a custom identified structure of MVP or HMVC instances with better overview due to the reduced set of design units or to adopt an easy to identify hierarchical structure of classic MVC [31] with small fine-grained triads that follow a stereotype assignment approach of GUI design units (every UI-Control potentially serves as a triad connected to a global *Model* or a part of it). It has to be considered that the classic MVC approach can only be relied on as far as the *Presentation* is concerned. A *Dialog Logic* or *DialogKernel* unit of design and their responsibilities cannot be covered and have to be realized by custom solutions. According to the HMVC or MVP approach, the *Controllers* couple the different triads for communication and navigation purposes, so that the evolution or maintenance of both *Presentation* and *Dialog Logic* or *DialogKernel* units of design is closely coupled. Finally, this approach needs a further separation of concerns to resolve the issue. A perfect distribution of responsibilities will be difficult to achieve, since there are only certain triad members to accept the set of responsibilities symbolized by the principal software categories *View Definition, View Navigation, Presentation Event Handling, Dialog Data Model* and *Dialog Event Handling*. These need to be distributed among the triad members.

3) *User Interface Patterns and Solution Approaches*

Before we draw our conclusions, we briefly discuss how the incorporation of UIPs for the *Presentation* component may directly or indirectly resolve some of the identified GUI design issues.

AT software. At first, the mixture of application and technical aspects can directly be avoided by the integration of UIPs. In this context, UIPs promise the reuse of visual layout and related interaction. Thereby, the stereotype parts therein would be implemented once and encapsulated in the UIP units. Then 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.

Event differentiation by software categories. To integrate UIPs in the *Presentation*, the differentiated software categories for event processing will be of great value. The differentiation of events is a fundamental preparation for UIP integration 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.

We favor a solution that corresponds to the responsibilities of the software category tree and identifies *Controller* like design units accordingly. In detail, we think about moving away from the concrete representation of visual elements in each *View* of any triad. *Controllers* on different *A* software levels should be established along with abstract to more concretely defined *View* contents: *Controllers* based on the *Presentation* sub-tree of software categories can be closer coupled to a *View*, than *Controllers* of the *Dialog Logic* sub-tree. For instance, for a *Dialog Logic* level based *Controller* a visibility could be defined that describes an associated *View* to be controlled in state with only abstract elements like inputs, outputs, commands and navigation signals (compare abstract user interface, abstract interaction components of reference [50]), since this level of detail is completely sufficient for this type of *Controller*. In addition, this design keeps the opportunity to easily change the concrete details of the concrete *Views* lower in hierarchy. The higher situated *Controllers* do not depend on the concrete details; as long as the number of view states and in- as well as output events remain the same, details of views concerning layout may freely be changed. *View* states will be relevant for the *Dialog Logic,* but not their concrete visual appearance. The *Dialog Logic* is decoupled and kept independent from *Views* in turn.

In common MVC architectures, the *Controllers* are closely coupled to the *View* they are associated with. When the *Views* are altered or exchanged, the *Controllers* need to be also adapted or will not be reusable at all. For UIPs, these circumstances are not desirable; some *Controller* tasks need to be stable and reusable, so that at least the design units controlling the presentation logic states remain unaffected.

The above described approach to a solution is exactly what UIPs may need: *Controllers* cannot rely on knowledge about the *View's* concrete visual composition, instead a small interface is required that is both used for communication between *Dialogs* and UIPs as units on *Presentation* level and for the configuration or instantiation of UIPs. The UIP just required to provide the states, in- and outputs of data required by the *Dialog Logic* part. Anyway, these requirements have to be met by any other *Presentation*, which may be not UIP based, in order to comply with the underlying use case. Therefore, we suggest that an abstract representation of the *Presentation* from the *Dialog Logic's* point of view is sufficient and are confident that this approach will improve software architecture quality.

UIP impacts. To conclude, the software category tree displays the dependencies among the occurring GUI responsibilities. When UIPs are to be integrated in the GUI software architecture, an architect is able to assess the impacts UIPs may have on the established relationships. In particular, he can decide what interactions require a different design for coupling in order to enable the reusability and exchangeability of UIPs. A first description of such assessments was presented in reference [51], but this was based on an earlier revision of the software category tree.

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. This display is independent of any platform, framework or architecture pattern. In contrast, available patterns can be be detailed or adapted on that basis. Eventually, the identified and described responsibilities can be re-structured in a GUI software architecture that may serve as a basis for a standardization of UIP integration. When no concern is mixed-up, reuse of UIPs is principally facilitated.

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.

Software categories and their relationship to patterns and design. One might ask what the difference is between the reviewed GUI architecture patterns with the presented tables of their responsibilities and the software category model, which nearly contains the same set of responsibilities.

Foremost, the software category model of course contains each responsibility of the patterns and is partly sourced from them. Nevertheless, the difference of capital importance is that the patterns already contain roles or design units with their fixed interfaces, dependencies and associations. These comprise the design as a structural and behavioral pattern unit and cannot be altered without changing the entire pattern concept.

On the contrary, the software categories model the responsibilities not from a fixed role perspective but from a point of view what topic, entities and operation types with their intended purpose are required for a certain responsibility. Hence, responsibilities in the software category tree are based on differentiated areas of knowledge and not on structural relationships in the first place. The advantage of the software categories is that they can be re-assigned to different designs, so that developers can be assured of completeness when each of the software categories can be traced to the resulting design. In that way, the same tasks the patterns serve are realized but different variations in design can be probed in a controllable manner. The patterns do not enable such a fine-grained decomposition of their responsibilities and allow no easy modifications without compromising the pattern' characteristic effects or forces.

Finally, the software categories do not only allow the allocation of responsibilities to designs; they are essentially supplemented with rules [6] that are to be applied on the design of interfaces between interacting entities. This concept of rules shall ensure an improvement of coupling and a reduction of dependencies.

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 [30]. 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 and reusable artifacts. The solution must resolve the identified GUI design issues to allow the integration of UIPs as artifacts that enable a reduction of efforts for the adaptation of GUIs. Finally, UIPs shall not add additional dependencies, otherwise they would make GUI software systems even more difficult to maintain.

The established GUI software category tree will help us to integrate UIPs into the existing responsibility relationships and keep control about their influence. However, the software category tree needs to be approved in practical applications and possibly requires a revision.

Acknowledgment

We like to express our gratitude to the companies, close friends and family members that took part in and contributed valuable results to the survey mentioned in the introduction.

References

[1] S. Wendler and D. Streitferdt, “A Software Category Model for Graphical User Interface Architectures,” The Ninth International Conference on Software Engineering Advances (ICSEA 14) IARIA, Oct. 2014, Xpert Publishing Services, pp. 123-133, ISBN: 978-1-61208-367-4.

[2] 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.

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

[4] S. Alpaev, “Applied MVC patterns. A pattern language,” The Computing Research Repository (CoRR), May 2006, http://arxiv.org/abs/cs/0605020, 2015.06.01.

[5] 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), 2015.06.01.

[6] M. Potel, “MVP: Model-View-Presenter. The taligent programming model for C++ and Java,” Taligent Inc., 1996, http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, 2015.06.01.

[7] A. Bower and B. McGlashan, “Twisting the triad. The evolution of the dolphin smalltalk MVP application framework,” Tutorial Paper for European Smalltalk User Group (ESUP), 2000, Object Arts Ltd., 2000, http://www.object-arts.com/downloads/papers/ twistingthetriad.pdf, 2015.06.01.

[8] J. Smith, “WPF Apps With The Model-View-ViewModel Design Pattern,” Microsoft Developer Magazine, 2009, Februrary, https://msdn.microsoft.com/en-us/magazine/dd419663.aspx, 2015.06.01.

[9] A. Ferrara, “Alternatives To MVC,”, http://blog.ircmaxell.com/2014/11/alternatives-to-mvc.html, 2015.06.01.

[10] 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.

[11] M. Scarpino, SWT/JFace in action. Greenwich: Manning, 2005.

[12] T. Hatton, SWT: a Developer’s Notebook. Beijing: O'Reilly, 2004.

[13] R. Steyer, Google Web Toolkit: Ajax-Applikationen mit Java. Unterhaching: entwickler.press, 2007.

[14] 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.

[15] M. Fowler, “GUI Architecures,” 18.07.2006, http://martinfowler.com/eaaDev/uiArchs.html, 2015.06.01.

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

[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] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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/.

[31] G. E. Krasner and S. T. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk 80,” Journal of Object Oriented Programming, vol. 1, August/September, 1988, pp. 26-49.

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

[33] D. Collins, Designing Object-Oriented User Interfaces. Redwood City, CA: Benjamin/Cummings Publ., 1995.

[34] E. Horn and T. Reinke, Softwarearchitektur und Softwarebauelemente [Software architecture and software construction elements]. München, Wien: Hanser, 2002.

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

[36] D. Greer, “Interactive Application Architecture Patterns,” http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/, 2015.06.01.

[37] S. Borini, “Understanding Model View Controller,” http://forthescience.org/books/modelviewcontroller/00_introduction/00_preface.html, 2015.06.01.

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

[39] Open Qusasar Sourceforge project, http://sourceforge.net/projects/openquasar/, 2015.06.01.

[40] B. Humm, “Technische Open Source Komponenten implementieren die Referenzarchitektur Quasar [Technical Open Source Components implement the Reference Architecutre of Quasar],” in: ISOS 2004 - Informationsysteme mit Open Source, H. Eirund, H. Jasper, O. Zukunft, Eds. Proceedings GI-Workshop, Gesellschaft für Informatik, 2004, pp. 77-87.

[41] B. Humm , U. Schreier, and J. Siedersleben, “Model-Driven development – hot spots in business information systems,” Proceedings of the First European conference on Model Driven Architecture: foundations and Applications, Springer LNCS 3748 , pp. 103-114.

[42] 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, IARIA, May 27 - June 1 2013, Xpert Publishing Services, pp. 34-43, ISSN: 2308-3557.

[43] 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.

[44] R. Stafford, “Service Layer,” in [32].

[45] F. Paternò, C. Mancini, and S. Meniconi, “ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models,” Proceedings of The Sixth International Conference on Human-Computer Interaction, INTERACT 1997, IFIP Advances in Information and Communication Technology, Springer, 1997, pp.362-369.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] E. Mbaki, J. Vanderdonckt, J. Guerrero, and M. Winckler , “Multi-level Dialog Modeling in Highly Interactive Web Interfaces,” The Seventh International Workshop on Web-Oriented Software Technologies (IWWOST 2008), ICWE 2008 Workshops, pp.38-43.

[51] S. Wendler and D. Streitferdt, “The Impact of User Interface Patterns on Software Architecture Quality,” The Ninth International Conference on Software Engineering Advances (ICSEA 14) IARIA, Oct. 2014, Xpert Publishing Services, pp. 134-143, ISBN: 978-1-61208-367-4.