Automated User Interface Generation Involving Field Classification

Matrtin Tomasek and Tomas Cerny

*Dept. of Computer Science, FEE Czech Technical University in Prague
Technicka 2 Prague, Czech Republic
E-mail: {tomasma5; tomas.cerny}@fel.cvut.cz*

Received 31 October 2016; Accepted 9 January 2017;
Publication 31 January 2017

Abstract

Software applications are designed with a concrete purpose in mind, specified by business owners basing on the individual requirements. The user-system interaction is specified in the analysis phase. This phase specifies inputs, outputs, interaction, etc. These elements could be specified separately based on the target platform. The designers know that the mobile clients could have a different application flow than desktop clients. However, this is not a rule and designer rarely think about the user context or application. This implies that outputs, inputs and interactions do not change automatically during the software lifecycle. In this paper we present techniques that can determine whether the user, which is in a particular context, should be required to spend his/her time to fill in fields that are not needed for accomplishing a specific business task. Moreover, these techniques are able to determine whether the fields should display or not, as well as how the system might interact with the user. Next, we present a computational architecture capable of these types of determinations and we demonstrate this technique in case study. Finally, we show how automatically combine user interface generation with our approach.

Keywords: MDD, automatically user interface generation, context-aware, business case, field classification, cross-cutting concerns.

1 Introduction

The purpose of user-oriented applications is processing user requests and responses whereby the user is interacting with the system. The interaction is described through a system sequence diagram during the analysis phase. The diagram is created based on the business system requirements. These requirements change during the software's lifecycle and in most cases they are not designed for the target device but rather for the application purpose. The term *RESPONSIVE* [13] describes software functioning in a way that is particular to different target devices. The device could be a mobile phone, a desktop computer, a notebook, a tablet, etc. Each device has a different resolution and when the software is responsive, the provided content displays properly on each device [13]. However, responsiveness in the design is not the only condition that affects the end-user's experience with each particular device. Moreover, this does not reflect the end-user's requirements, while using each particular device, nor the different situations in which the device is being used.

Users interact with the application using User Interface (UI). The UI allows users the ability to control the system and provides them with feedback representing the actual state. Next, it follows the user actions and requests [14]. In most cases the application collects data from users.

In this research, we focus on the input data that must be filled in by user application context. A common application knows which data the user is supposed to provide and displays the field that must be provided by the user. It is not designed to consider details about a user him/herself. Moreover, when the system displays more inputs than it is necessary, the human-error probability can increase [15]. This also applies when the UI contains a hidden field that is supposed to be visible. It could lead the process to fail, thereby causing problems to the user [15]. Therefore, the UI has *IF CONDITIONS*, which determines whether the field is supposed to be displayed or not. These conditions can base on the user's roles in the system, or on the selected value in the fields, or on the application state. However, this is hard-coded in a particular page in most cases, whereby every change is error-prone and the code insight is much harder, deteriorating readability. Such approach is time inefficient and hard to maintain.

The process, therefore, should be improved in such a way that the system knows which fields are displayed based on the context. This knowledge can improve readability and maintainability. The goal of this paper is present an approach that determines whether a data field should be rendered and

displayed based on a particular context. The approach is user-oriented and its aim is to design solutions helping users accomplish their goals. Next, we classify form fields and apply this classification in order to determine how to integrate these fields into the presentation. Furthermore, it offers an appropriate validation strategy.

The UI is designed for users to help them accomplish their goals. It is of critical importance to know what information the user must provide and what information is not necessary. This reduces the time required to accomplish a task within the system by a particular user. Next, it increases the probability that the user finishes the task and avoids interruption. Finishing the process is the core purpose of any designed application.

This paper is organized as follows: The next section describes related work, followed by description of proposed solution and case study section. Finally, an integration between proposed solution and automatic user generation process is presented with conclusion.

2 Related Work

The UI creation process involves multiple phases that focused on different dimensions. The early phases do not involve the UI itself. First of all, we have to create use-cases and then we can identify processes that are critical to accomplish expected tasks. Each such a task can be divided into multiple subtasks and each subtask can be representing somehow in the UI. The Task Model [2] could help us to track tasks dependencies and their flow. However, the task itself does not create the UI. Some contemporary approaches based on a created model that represents the UI and they expect the developer to work with this model. These approaches that derive UIs from a model involve Model-Driven Development (MDD) [3, 12]. When such a model changes, the UI derived from it should reflect the change as well. This approach improves maintainability and it makes the software model easy to comprehend and easy to read, especially to analysts. The UI is derived from the model where changes directly apply to the UI in the MDD approach. The model helps us to keep an application consistency and complexity [16]. The final view that is displayed to the client is determined based on the model. There is also needed a controller that takes care of user's actions. A common pattern used is called Model-View-Controller (MVC) [10], which bases on concern separation. The User Interface could be also generated based on transformation a model into a metamodel [16] with the respecting MVC pattern. The view itself could use the controller to decide whenever is field displayed or needed. However,

the MDD does not reflect the importance and the field severity in different user-context and in different business phase. If this type of functionality is required, then it must be implemented for the concrete purpose.

The UI specification is, however, developed by yet another person who may have different expertise [6]. There are multiple parties involved in the process including various stakeholders, User Experience (UX) teams, software developers, etc. This process could be improved in the event that the UI is presented to other people working in differing contexts [6], however it requires many people working on the model, which is subsequently transformed by the software developer. The advantage is that all of the various roles working on the project contribute elements from various fields of expertise. This process focuses on the static UI, but does not focus on the UI that would change at runtime. The result of this process is a suitable UI with multiple *IF CONDITIONS* that produces different UI components to different users. The component is derived from the model and therefore we need different models in different situation. The process could be improved when the situation and its dependency on the model are described in runtime.

The problem with MDD is that it does not generate the UI automatically whereby someone has to change the UI part. Moreover, the MDD does not deal properly with various particular aspects of the software, such as the user context, the security context, the application context, etc [3]. Essentially, the UI is comprised of several crosscutting concerns [19]. We could use Aspect-Oriented Programming (AOP) [9] to create dynamic views. Most of the AOP-based UI approaches can produce the view basing on context and not deteriorate readability. The context may be the security context, the user context, and/or the application context. The AOP approach is able to decide whether a field should display or not. When the automated user generation applies [3] then the UI can reflect changes in the model directly. Moreover, it reduces the number of errors [4] produced during the entire software lifecycle. It brings time reductions and improves software maintainability, since every change in the model directly propagates and reflects in the UI. We can also improve process with adaptive application structure [17] that helps with application flow.

The AspectFaces approach [1] involves AOP, and creates a dynamic view based on user context. This approach involves code inspection that interprets templates in order to create particular UI fragments representing data. The decision as to whether the field is displayed or not bases on user context and conditions specified in the template. This implies that when the condition changes, the template must change as well. The UI fragment could

be created as an HTML page [8] or it could be created as a platform of independent definitions [18] that processes on the client-side in the client-server architecture. The AFRest [18] approach extends to the AspectFaces approach and brings platform independence through the REST web services producing JSON [5]. This approach deals with dynamic objects and enables to create views based on the model. All the UI field properties are available in the provided context for possible use on given platform. Therefore, the client-side that creates the UI can determine the work with the UI fragment. The server does not consider any target client platform. It could be an Angular, Android, Windows Phone or other type of a client.

End-users need a variety of views depending on a particular situation [3], therefore we need a dynamic view. The dynamic configuration [7] could be specified in the form of XML as generalized rules and it can affect final UI. We are able to describe custom localization and behavior. We need considerable number of dynamic configurations to cover all possible scenarios. The other possibility is to use a pattern based UI generator [21] to create a UI. This generator works with context, model and patterns. This approach creates a platform Independent User Interface Model (PIUI) [21] and it applies the UI mapping to create a Platform Dependent User Interface Model (PDUI) [21]. This approach is similar to the AOP approach.

The MDD and AOP approaches itself are not designed to collect all business requirements and reflect them into target application. When the UI should be different for different type of users, then we have to focus on transformation and transformation mapping that transform model into final view. The adaptive user interface could be proposed based on the parametrized MDD-transformation [22] and user context. The process itself involves multiple transformation of current model into the final model [22]. The MDD also is not able to work with a user context. If we would like to reflect the user's context like slow internet connection, then we have to also extend the MDD approach. We could use transformation rules [20] and transform model into another model. However, these techniques do not reflect the business requirements, because these approaches do not have any information about the purpose and severity of the UI field that must be rendered. The presented approaches cannot specify how the system should work with a particular field, within the context of a particular user situation. This is because they are not designed to capture their meaning, in the context of completing a user request. There can be conditionals specifying whether the field is displayed or not, but this approach cannot handle large numbers of possibilities, therefore it negatively impacts maintainability, readability and lacks separation of concerns.

The process could be improved when divided on two phases. The first phase specifies the field meaning and decide whether the field is displayed in the particular user-context. The second phase propagates the information into the generated view. This approach centralizes the information about fields into a single place and it separates process into two independent processes, that could work separately. Next, it reduces a business logic required to customize the view for different users in different user-context.

3 Proposed Solution

The mechanism classifying UI fields in the business case is divided on two sections. The first section represents the main concept and provides a model that qualifies UI fields. The second section demonstrates a computational model that is able to classify UI fields.

3.1 Main Idea

The application displays UI fields that are required to be provided in by user if the user wants to finish the process, e.g. booking a flight ticket. The process, in this case, ends with a ticket reservation. Payment of the ticket is not considered in this example. The class diagram is provided on Figure 1. This is a very simplified part of an online booking system for an airport. If a user wants to create a reservation he/she has to choose a flight, date, and the total number of tickets on that flight. We also need some information about the user. If the user him/herself is not considered, the system is fairly straightforward. The application and the security context is excluded and the key focus becomes the particular user context. There are several indicators that provide information about the user and user's particular circumstances. The first indicator is the user device. The system should work differently when these is a desktop computer and when he or she is working on a mobile phone. In the second case, there are data available only when user requests them. If the user wants to book a ticket on a flight with a departure time in two hours, we can expect that the user is in a time press. When the user wants to book a ticket on a flight with a departure time in ten days, then we can expect that the user consider making a reservation. Other situations are considered later in the case study section.

3.2 Classification Model

When the system decides how to work with the user, the behavior must be specified. We created the classification model that is able to hold all necessary

Figure 1 Classification model.

information. The classification model is presented in the Figure 1 and it is detailly described in this section. We represent the model as UML class diagram with analytics classes with full relationships but without method. The UI bases on business cases in which we can create a specific business case and associate it with a particular field. Each particular business case has multiple phases. The Business Case Phase (BCP) is associated with fields that are relevant in this phase. The business case field has its own level of severity (BCFieldSeverity).

The *BCFieldSeverity* holds information about the field importance. Basic levels of severity were created. The field importance is an enumeration class that could be extend if it is needed. The class with all relationship is in the Figure 1. Indication of a *CRITICAL* enumeration value in the Severity field implies that without this field it is impossible to accomplish the task. Indication of a *REQUIRED* enumeration value in the *SEVERITY* indicates that this value is not critical for accomplishing the task, however it helps to define the user or improve the user's process while using this software, which could lead to accomplishing the task. Indication of a needed enumeration value in the *SEVERITY* indicates that this value is needed only for the customization of the UI. Indication of a *NICE.TO.HAVE* enumeration value in the Severity is used only for inputs that could be used at a later time, but these values are not mandatory. The *BCFieldSeverity* in the Figure 1. has the purpose of a particular field.

The *PURPOSE* holds information about the field purpose in the system. The relationship with all possible values is in the Figure 1. Indication of the *SYSTEM_IDENTIFICATION* enumeration value in the *PURPOSE* field indicates that the value is necessary to identify and complete the task in the system. Indication of the *SYSTEM_INFORMATION* enumeration value in the *PURPOSE* field indicates that the value is collected and it is used by the system. Indication of the *INFORMATION_MINING* enumeration value in the *PURPOSE* field suggests that the information is used for data mining and it can provide beneficial information in the long term. The future interaction indicates that these values are used only to improve the interaction between the system and the user.

The computational model presented in Figure 2, performs all computations and it calculates the severity score (the model is described in the next subsection). The score is the input value, and based on this, the system decides how to interact with users in the particular field within a concrete business case. The BCP is associated with a configuration. The configuration applies to all fields in the *BCP* and provides a mechanism to influence the result

Figure 2 Computational model unit.

of the process. The configuration specifies the threshold for different types of behavior. It is described in the *BEHAVIOR* entity. Indication of the *REQUIRED* value in the *BEHAVIOR* enumeration indicates that the value is required and is also validated by all validators present on the field. Indication of the *VALIDATION* value in the *BEHAVIOR* indicates that the value is not required; when it is present the value is validated. Indication of the *ONLY_DISPLAY* value in the *BEHAVIOR* indicates that the field is displayed, but it is not validated even in the case that the value has been filled in. Indication of the *HIDDEN* value in the *BEHAVIOR* indicates that the field is present in the UI, but is hidden. Indication of the *NOT_PRESENT* value in the *BEHAVIOR* indicates that the field will not display to the user.

The business case and its dependency on the system were described, although we did not focus on the user, who uses the system, in this phase. The user is referred to as ‘client’ in the model. The client is specified by the device, which is used to control the system and with various client properties, which contain detailed information about the client. The solution works with the connection type, the current battery capacity, the display resolution, and the position. This information determines the state of the client and the computational module, which is described in the next section, altogether creating the *GeneratedField* entity that contains information about the field classification. This entity holds information about how the system should work with this field based on the user context, the business case, and the application context.

3.3 Computational Module

The model is used to represent the main concept and to transfer information to the target system. Next, it contains information about user and transfers the user information to the Computational Module Unit (CMU). The *CMU* computes the score for the field in each concrete business case. It also builds the *GeneratedField* entity.

The illustration is provided on Figure 2. The *CMU* contains two units: The Scoring Computational Unit (SCU) and the Classification Computational Unit (CCU). We use the pipe and filter architecture; therefore the output from *SCU* is the input for the *CCU*. The SCU computes the score and stores it in the *BCFieldSeverity* entity. The SCU needs the level of severity, the purpose, the actual value and the set of possible values, if they exist. The severity and the purpose enumerations were described in the previous section. The two remaining properties are described in this section.

The actual value represents the actual field value. This value is displayed to the client. The value can affect a lot of things for example the different fields could be displayed based on this property. The possible values are all values that can be in this field. The usual example is a drop down menu. The result (*score*) is computed based on these input fields. The details of this process are outside of the scope of this paper, but an equation could be created based on the target system requirements. It is easy to maintain a computation equation in one place whereby the maintenance of the *IF CONDITIONS* are hardcoded in the source code and could be duplicated in a different place inside of the application.

The (*SCU*) computes the fields score. The result does not reflect the state of the field because the state is created in the *CCU*. The input is the score from the previous step. The *CCU* also needs information about the client's context. It needs to know which device is used as well as the user's properties. The last input is the *configuration*, provided by the software developer. This *configuration* has a threshold of different types of behavior. The output is *GeneratedField* entity holds information about how the system should work with the field. Upon configuration changes the UI is affected.

4 Case Study

An airport reservation system is used as a case study to demonstrate the presented technique. The simplified schema is presented on Figure 3. The two use cases are applied to describe different situations to demonstrate the usage of the proposed solution.

The first step is that the business owner has to create a business case. The business case is making a complete ticket reservation, which generates revenue for the company. This business case is divided into 2 phases in this case study. The first is searching for a flight. The second is making the reservation of a flight ticket. Ticket payment is not considering as this is out of the scope of this paper. The first phase is referred as the 'Search flight' and the second is referred as the 'Reservation flight'.

Figure 3 Case study model – flight ticket reservation system.**Table 1** Business case field definition for search flight. S_ = SYSTEM and F_ = FUTURE

Field	Severity	Reason
source	CRITICAL	S_IDENTIFICATION
destination	CRITICAL	S_IDENTIFICATION
start	REQUIRED	S_IDENTIFICATION
end	NEEDED	S_IDENTIFICATION
price	NEEDED	S_INFORMATION
rank	NICE_TO_HAVE	F_INTERACTION

Table 1 provides setting created by software developer and contains information about the fields necessary to search for a flight. Table 2 shows the configuration of both business cases.

4.1 A Desktop Client

The first use case is as follows. The user wants to reserve a flight ticket from Prague to London on his trip across the Europe. The user context is input to *SCU* too. The information is that the user is working on a desktop computer as well as information that user is connected via a cable Internet connection. The *SCU* computes the *Score* value, which equals 100 for the source, destination

Table 2 Configuration used for CCU

Range	Behavior
90–100	REQUIRED
60–90	VALIDATION
40–60	ONLY_DISPLAY
10–40	HIDDEN
0–10	NOT_PRESENT

and start fields, 62 for the end and price field, 50 for the field rank in this example, this calculation gives Table 3. The end result is the input to the CCU. The other configuration input shows Table 2.

The result is that only the *RANK* field displays. The *PRICE* is not required, but it is displayed and validated when a user fill in this field. The same applies for *END*. The *SOURCE*, *DESTINATION*, *START* are displayed and must be provided in by the user. The result is presented in Table 3.

The second part is a ticket reservation. The business owner has to create business phase for this part. It is presented on Table 4. The critical fields are *PHONENUMBER*, *EMAIL* and *NUMBER*. These fields are necessary to send a ticket to a user and to calculate total price. We use the same configuration for both use cases. The client is also the same. It is the user who works on notebook and is connected via LAN.

Table 3 Classified field for search flight for desktop and mobile client

Field	SCU Desktop	SCU Mobile	CCU Desktop	CCU Mobile
source	100	100	REQUIRED	REQUIRED
destination	100	100	REQUIRED	REQUIRED
start	100	86	REQUIRED	VALIDATION
end	62	42	VALIDATION	ONLY_DISPLAY
price	62	42	VALIDATION	ONLY_DISPLAY
rank	50	50	ONLY_DISPLAY	HIDDEN

Table 4 Business case field definition for reserve flight. S_ = SYSTEM and F_ = FUTURE

Field	Severity	Reason
name	NEEDED	S_INFORMATION
surname	NEEDED	S_INFORMATION
phoneNumber	CRITICAL	S_IDENTIFICATION
email	CRITICAL	S_IDENTIFICATION
street	NICE_TO_HAVE	F_INTERACTION
city	NICE_TO_HAVE	S_INFORMATION
number	CRITICAL	S_IDENTIFICATION

Table 5 Classified field for reservation flight for desktop and mobile client

Field	SCU Desktop	SCU Mobile	CCU Desktop	CCU Mobile
name	62	42	VALIDATION	ONLY_DISPLAY
surname	62	42	VALIDATION	ONLY_DISPLAY
phoneNumber	100	100	REQUIRED	REQUIRED
email	100	100	REQUIRED	REQUIRED
street	50	30	ONLY_DISPLAY	HIDDEN
city	55	35	ONLY_DISPLAY	HIDDEN
number	100	100	REQUIRED	REQUIRED

The *SCM* computes the score for field. The score for *NAME* and *SURNAME* is 62. The field *STREET* has score 55. Similar score is for city, 50. The fields *PHONENUMBER*, *EMAIL* and *NUMBER* have the score equal to 100. The *CCU* results with displaying the fields *NAME* and *SURNAME*. These are validated only when user fills in these fields. Fields *STREET* and *CITY* are only displayed. The fields *PHONENUMBER*, *EMAIL* and *NUMBER* of tickets are all displayed, validated and required. The result is presented in Table 5.

4.2 A Mobile Client

The second use case is as follows. The user wants to reserve a flight ticket from Prague to London on a business trip for the following day. The user is working on a mobile phone with a 3G internet connection. The *SCU* computes different score, because the client has changed, therefore the *CCU* produces different results. Moreover, we can use different *configuration* to modify entire *CCU* output. The *SOURCE* and the *DESTINATION* fields are the same because these are system requirements regardless of the situation. The field *START* is validated when filled. The fields *END* and *PRICE* are only displayed. This is due to the fact that the client is moving and he/she is working from a mobile device so the system assumes that the user might not be willing to provide these fields. The *RANK* is hidden in the view. The system makes an assumption that the airport rank is not relevant in this particular situation. The results are presented in Table 3.

The second part is a ticket reservation. The client remains the same. It is person who uses mobile device and who is connected via 3G-connection type. The *SCU* unit computes different score than in the first case, when the user used notebook. The score for *NAME* and *SURNAME* is 42. These fields are only displayed. The fields *CITY* and *STREET* are hidden and they have score 35 and 30. The fields *PHONENUMBER*, *EMAIL* and *NUMBER* are required and they have to be filled in. The system decided that the information about user name and surname are not necessary to complete reservation. The key

information is contact information, because the ticket must be delivered to user. The system also needs information about the number of ticket which users pays. It is critical to calculate price. The result is presented in Table 5.

The UI is presented to the user adapts basing on client's context and the business case specification. It has changes basing on the configuration and the computation module. The changes are performed automatically and they do not need to result in any manual changes in the source code of the UI.

4.3 Threats to Validity

The measurement for case study was generated on the same computer at the same time. The results were derived from case study model. We simplified the model, because we wanted to demonstrate the proposed solution, but we did not want to design an airport reservation system. The model will be greater in the real-world usage. The client context was transferred to our proposed solution as static part. We did not use real device, but we simulated it. The Classification model was transformed into classes and implemented in the Java language. The Computational was also implemented in the Java language. We used the Java version 1.8, however does not have dependency on the Java version.

5 Integration with Automated User Generation Process

In this section we do focus on the connection between categorization and automatically generated user interface. The modern concept divides software into two parts. The first part is backend service, that implements business logic and it is used as a data layer. The second part is a frontend that is used to control an application. This division implies that if the backend service changes data layer then the frontend is changed too, because the frontend represents data in a business service and it is used to create new data in backend. The data are created in backend mostly via forms that are filled-in by a user. The form is created based on a model that holds a field that must be fill-in. It also holds information about data types, component order and etc. The model has not information about security context or user context.

We focus on the AOP-based UI method that uses code-inspection to generate UI. The output of this process could be useful UI or only a UI definition that is interpreted on target device. If the process creates UI definition [18], then this definition could be modified and it could reflect security context, user context or categorization context. We can combine cross-cutting concerns to create complete UI. The definition could be also

transferred to client and then executed. Therefore, this definition should be platform independent, because we not known where it will be used. The client could request UI definition whenever it is necessary. This approach has many advantages. It is able to reflect changes in the model in runtime. It is also able to reflect validations rules, component order, data consistency, user and security context. This approach allows control the UI on a target device in a runtime, therefore we can change the UI without publishing a new version for clients. The process itself is divided into two phases. In the first phase the client requests a UI definition and a server creates it for the client. The client sends to the server also all information about his context like type of connection, battery status etc. In the second phase the client receive a definition and creates an UI in a runtime based on a received definition. Our proposed solution computes a score for each field and the system decides how to work with this field based on the score and selected configuration. The UI generation process itself is not connected with the field classification. The integration between our proposed solution and the Aspect Faces REST (AFREST) [18] is done on the server side. When we change a configuration or a severity of the field, then our proposed solutions produces different output and this output is given as input to AFREST. The result of this action is that the AFREST produces different definition therefore the client creates the different UI. The client creates the UI based on UI definition that receives from the server, therefore we have to integrate our proposed solution on the server side. The changes on the client side are not necessary because the client side reflects changes in UI definition automatically.

5.1 AFRest with Field Classification Enrichment

So far, we explained how to categorize the UI field in a particular user context, but we did not focus on an automatically generated user interface. Next, we present our preliminary research on using AFRest [18] approach to obtain the UI definition. This definition is transferred on client side that executes it. The example of the UI definition is in Listings 1 and 2. Next, we present how the output from SCM and CCM affects the UI definition. It is presented in Listings 3, 4, 5 and 6.

We use our case study model to create a UI definition. We focus on the second phase that was a ticket reservation. Moreover, we consider only on the step when the user has to insert all necessary information and we choose two fields (surname and phoneNumber) to demonstrate the process. The definition without categorization is presented in Listing 1 and 2. The phoneNumber is text field, because the input value could contain special characters like + /.

The field is not read only it is visible and required. The number of minimal characters that must be insert is equal to 5. The field has no option value and it displays in one row with label before input box. The surname is also text field. It is not read only it displays and it is mandatory. The number of minimal characters that must be filled in is equal to 5 and the number of maximal characters that could be insert is equal to 255.

All clients receive the UI definition that is described in Listing 1 and 2 no matter if its a mobile client or a desktop client. The mobile client requires all validations and rules as a desktop client. If we want to involve the results from categorization process, then it must be done on the server side. The target device should not care about how the final definition was created. The UI definition will be created on the server side and then it will be changed based on the result of the categorization process and the final UI definition will be created. This approach allows current the UI generation process untouched and it allows changing the final view based on the output of categorization

Listing 1 Basic definition for field PhoneNumber

```
{
  "widgetType": "TEXTFIELD",
  "id": "phoneNumber",
  "label": "Phone Number",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
    {
      "validationType": "REQUIRED",
      "value": "true"
    },
    {
      "validationType": "MINLENGTH",
      "value": "5"
    }
  ],
  "options": null
}
```

Listing 2 Basic definition for field Surname

```

{
  "widgetType": "TEXTFIELD",
  "id": "surname",
  "label": "user.surname",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
    {
      "validationType": "REQUIRED",
      "value": "true"
    },
    {
      "validationType": "MINLENGTH",
      "value": "5"
    },
    {
      "validationType": "MAXLENGTH",
      "value": "255"
    }
  ],
  "options": null
}

```

process. We might change business case phase, configuration or fields on the server and the changes will be distributed to all clients immediately.

We used the regular definition that is presented in Listing 1 and 2 and we apply on it the output from categorization process. The final definition for desktop client is presented in Listing 3 and 4. The phone Number field remains untouched because the field is critical and it is used for a system identification purpose. The surname field has been changed. The required validation was removed, but other validations remains untouched. It has maximum length 255 and minimum length 5. These validations will apply only if a field is filled-in by user.

The final definition for mobile client is presented in Listing 5 and 6. The behavior for phoneNumber field is the same as for desktop client, but the surname field has been changed. This field has no validations but it is displayed to a user. The user can fill-in this field by any value or skip this field.

Listing 3 PhoneNumber definition for desktop client

```

{
  "widgetType": "TEXTFIELD",
  "id": "phoneNumber",
  "label": "Phone Number",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
    {
      "validationType": "REQUIRED",
      "value": "true"
    },
    {
      "validationType": "MINLENGTH",
      "value": "5"
    }
  ],
  "options": null
}

```

Listing 4 Surname definition for desktop client

```

{
  "widgetType": "TEXTFIELD",
  "id": "surname",
  "label": "user.surname",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
    {
      "validationType": "MINLENGTH",
      "value": "5"
    },
    {
      "validationType": "MAXLENGTH",
      "value": "255"
    }
  ],
  "options": null
}

```

Listing 5 PhoneNumber definition for mobile client

```

{
  "widgetType": "TEXTFIELD",
  "id": "phoneNumber",
  "label": "Phone Number",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
    {
      "validationType": "REQUIRED",
      "value": "true"
    },
    {
      "validationType": "MINLENGTH",
      "value": "5"
    }
  ],
  "options": null
}

```

Listing 6 Surname definition for mobile client

```

{
  "widgetType": "TEXTFIELD",
  "id": "surname",
  "label": "user.surname",
  "classType": false,
  "readOnly": false,
  "visible": true,
  "layout": {
    "layoutDefinition": "ONECOLUMNLAYOUT",
    "layoutOrientation": "AXISY",
    "labelPosstion": "BEFORE"
  },
  "rules": [
  ],
  "options": null
}

```

5.2 Threats to Validity

The measurement for second part of case study was generated on the same computer at the same time. The second part of our case study demonstrates

integration between the proposed solution and the AFRest. The integration respects the AFRest solution and it is written in the Java language. The UI definitions were created on the Glassfish 4.0 server with one deployed application that was AFServer application. This is a showcase application that demonstrates context-aware remote automatically user interface generation. The definition does not create UI, but it is send to target device that interpret this UI definition.

5.3 Summary and Comparison

In this case study, we present the usage of our solution. First of all, the business process is specified and it is split into business phases. Each of business phase has a direct connection with fields that a system requires. Moreover, the business phase specifies the meaning of fields and their purpose. Next, our solution consumes the user-context and it calculates the score for each field. The final decision, when the field must be displayed or not, is affected by configuration, that could be different for each business phase. These calculations and decisions are centralized in one place. It helps to keep the meaning and purpose together. It also improves the software maintainability. The centralization helps to improve future development and it increases the readability.

In this section, we compare our solution with a JavaServer Faces (JSF) [11] approach. The JSF separate the Logic Layer and the Presentation layer and it implements the MVC [10] pattern. JSF is a part of Java EE distribution and uses a XHTML page to render a view. The XHTML page could be created from multiple UI fragments and these fragments can use multiple controllers to control user's actions. The rendering of a particular field can be controlled in the UI fragment itself or in the controller. The first possibility is not recommended, because there should not be the logic in view. Moreover, we might have to duplicate this logic, because the field could be in multiple UI fragments. Therefore, we focus on the second possibility, because the second approach is much more familiar with our approach. The controller has a function that decides whether the field is rendered or not. This information is propagated to the view. We consider the search flight process and the example of search form in XHTML is in Listing 7. The search XHTML page has its own controller called searchController, it is in Listing 8. We set up required fields in the XHTML directly, because we know that these fields must be displayed and we do not add logic inside the view. The main parameters are rendered and required. The non-mandatory fields call two functions. The first function

Listing 7 Basic definition for field Surname

```

<h:form id="searchForm">
  <p:inputText value="#{searchController.source}"
    rendered="true" required="true"/>
  <p:inputText value="#{searchController.destination}"
    rendered="true" required="true" />
  <p:inputText value="#{searchController.start}"
    rendered="true" required="#{searchController.reqSource}"/>
  <p:inputText value="#{searchController.end}"
    rendered="true" required="#{searchController.reqEnd}" />
  <p:inputText value="#{searchController.price}"
    rendered="#{searchController.showPrice}" />
  <p:inputText value="#{searchController.rank}"
    rendered="#{searchController.showRank}" />
  <p:commandButton value="Submit" actionListener="#{searchController.reserve}"/>
</h:form>

```

Listing 8 Basic definition for field Surname

```

@ManagedBean(name = "searchController")
@RequestScoped
public class SearchController{
  Client client;
  public boolean reqStart(){
    boolean isFieldRequired = true
    if(client!=null){
      switch (client.getDevice()){
        case "PHONE":
          if(client.getConnectionType.equals("3G")){
            isFieldRequired = false;
          }
          else if(client.getConnectionType.equals("WI-FI")){
            isFieldRequired = false;
          }
          break;
        case "NOTEBOOK"
          isFieldRequired = true;
      }
    }
    return isFieldRequired;
  }
}

```

decides whether the field is displayed and the second function decide whether the field is validated and required. These functions are implemented in the search controller in Listings 8. We show example of required function.

Our approach holds information about a field purpose and severity in a one place. Moreover, it connects fields with phases and the computational model produces results for different clients in different contexts. If the field severity or purpose is changed, then our approach generates different outputs. The JSF approach does not hold the field purpose and severity information. The only part where is specified whether the view should display the particular field or not is in controller, and it is implemented as function. The function has limited

possibilities and we have to write for each field this type of function. When the field severity changes, then the function must be re-written. Moreover, the view contains small part of logic. We consider the true or false values as the small part of logic.

6 Conclusion

This paper presents an approach to specify UI behavior basing on the particular user context. The approach centralizes field specification enforcing rendering and required value requesting. It provides automated UI change propagation, of both the client and data structure changes. This approach can be used on its own or with an existing approach, i.e. AspectFaces or AFRest. Both approaches based on MDD to generate the view at runtime. The approach is a two-stage solution containing the model and the computational units. The static configuration setting and the business case specification are combined with a dynamic client context and with a system state to generate the entity. This entity contains information about how the field should be interpreted for various clients. The proposed solution is demonstrated in a case study wherein two different use cases with different clients were applied. The case study section also demonstrates the integration between the presents approach and the AFRest. This type of integration allows us to change a UI during runtime on all types of devices.

The approach benefits are business case centralization and field centralization. It decreases the number of *IF CONDITIONS* in the source code in view. Finally, it improves maintainance due to centralization of all factors to a single place.

We would like to demonstrate the usage of presented approach in the real-world application with much more number of use-case then we presented in the case study section. The selected application must be analyzed and we have to find out business cases that will be used as an input into our proposed solution. We also described the connection between classification and automatic user interface generation process. This connection should be improved and we would like to support more solutions to exceed the usage in the real-world. We expect that the real-world usage could test correctness and reliability of the proposed solution. So far, we did not discuss how to test the solution. However, the proposed solution contains unit, that produces result, therefore the test could be divided into two parts. The first part could test the output from SCU and CCU the units are in the Figure 2. The black-box test could be used to verify the results. The second part could test the correctness of generated UI. The UI tests like Selenium test could be used.

Acknowledgment

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/234/OHK3/3T/13.

References

- [1] Aspect Faces (2015). Available at: <http://AspectFaces.com>
- [2] Akiki, P. A., Bandara, A. K., and Yu, Y. (2016). Engineering adaptive model-driven user interfaces. *IEEE Transact. Softw. Eng.* 42, 1118–1147.
- [3] Cerny, T., Cemus, K., Donahoo, M. J. and Song, E. (2013). Aspect-driven, data-reflective and context-aware user interfaces design. *SIGAPP Appl. Comput. Rev.* 13, 53–66.
- [4] Cerny, T., and Song, E. (2012). Model-driven rich form generation. *Informat. Int. Interdiscipl. J.* 2012, 2695–2714.
- [5] Fredrich, T. (2015). *RESTful Service Best Practices Recommendations for Creating Web Services*. Available at: [http://www.restapitutorial.com/media/RESTful Best Practicesv11. pdf](http://www.restapitutorial.com/media/RESTful%20Best%20Practicesv11.pdf)
- [6] Garcia, A., Frey, J., Sottet, S., and Vagner, A. (2015). “A multi-viewpoint approach to support collaborative user interface generation,” in *Proceedings of the Computer Supported Cooperative Work in Design (CSCWD), 2015 IEEE 19th International Conference, Slovakia* 425–430. 2015.
- [7] Guetari, R. (2003). “Xml as a dynamic ui and database access customization mean” in *Proceedings of the Systems, Man and Cybernetics, 2003. IEEE International Conference, Rome*, 4840–4845.
- [8] Hickson, B. R., and collective. (2015). *Html 5.1 w3c Working Draft 17*. Available at: <https://www.w3.org/TR/html/>
- [9] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J -M. Videira, C., Maeda, C., and Mendhekar, A. (1997). “Aspect-oriented programming,” in *Proceedings of the IECOOP’97-Object-Oriented Programming, 11th European Conference, Berlin: Springer*, 220–242.
- [10] Leff, A., and Rayfield, J. T. (2001). “Web-application development using the model/view/controller design pattern,” in *Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference, Vienna*, 118–127.
- [11] Li, S., and Sun, L. (2011). “Advantages analysis of jsf technology based on j2ee,” in *Proceedings of the 2011 International Conference on Computer Science and Service System (CSSS), Bangkok*, 2008–2010.
- [12] Macik, M., Cerny, T., and Slavik, P. (2014). Context-sensitive, cross-platform user interface generation. *J. Mult. User Interfaces* 8, 217–229.

- [13] Mohorovicic, S. (2013). "Implementing responsive web design for enhanced web presence," in *Proceedings of the Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention, Opatija*, 1206–1210.
- [14] Norman, D. A. (2002). *The Design of Everyday Things*. New York, NY: Basic Books.
- [15] Reeder, R. W., and Maxion, R. A. (2005). "User interface dependability through goal-error prevention," in *Proceedings of the International Conference on Dependable Systems and Networks*, Anchorage, AK, 60–69.
- [16] Roubi, S., Erramdani, M., and Mbarki, S. (2016). "Modeling and generating graphical user interface for mvc rich internet application using a model driven approach," in *Proceedings of the 2016 International Conference on Information Technology for Organizations Development (IT4OD)*, Morocco, 1–6.
- [17] Sebek, J., and Richta, K. (2016). *Usage of Aspect-Oriented Programming in Adaptive Application Structure* Cham: Springer International Publishing, 217–222.
- [18] Tomasek, M., and Cerny, T. (2015). "On web services ui in user interface generation in standalone applications," in *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems, RACS*, New York, NY, 363–368.
- [19] Verma, P. K., Dahiya, D. and Jain, P. (2010). "Using aspect oriented software architecture for enterprise systems development," in *Proceedings of the Fifth International Conference on Digital Information Management (ICDIM)*, Portugal, 448–453.
- [20] Yaici, K., and Kondo, K. (2008). "A model-based approach for the generation of adaptive user interfaces on portable devices," in *Proceedings of the 2008 IEEE International Symposium on Wireless Communication Systems*, Rome, 164–167.
- [21] Zhang, L., Gong, B., and Liu, S. (2008). "Pattern based user interface generation in pervasive computing," in *Proceedings of the Third International Conference on Pervasive Computing and Applications, ICPCA* Orlando, FL, 48–53.
- [22] Zouhaier, L., Bendaly, Y., and Ben Ayed, L. J. (2015). "A model driven approach for improving the generation of accessible user interfaces," in *2015 10th International Joint Conference on Software Technologies (ICSOFTE)*, France, 1–6.

Biographies

M. Tomasek received his Bachelors and Engineers degrees from the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. He is Ph.D. student in Czech Technical University in Prague. His area of interests is software engineering and enterprise application development. His main research include automatic user interface generation process, networks and service oriented applications.

T. Cerny received his Bachelors and Engineers degrees from the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. Next, he received his M.S. degree from Baylor University and in 2016 finished Ph.D. in Information Science and Computer Engineering. Since 2009, he works as an Assistant Professor at FEE, CTU. His area of research is software engineering, separation of concerns, model-driven development, enterprise application development and networking.