TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics

Jevgeni Tolstouhhov

Conceptual analysis and development of the user interface framework

MASTER THESIS
IDX70LT

Supervisor: Professor Jaak Tepandi

Tallinn 2008

This work is composed by myself independently. All other authors' works, essential states from literary sources and facts from other origins, which were used during the composition of this work, are referenced.

Jevgeni Tolstouhhov

12.05.2008

Annotation

Current work introduces conceptual approaches for implementing a user interface framework of Customs Engine platform developed at Cybernetica AS by the author of this thesis. This thesis embraces the complete evolutionary process from the very first version of the framework to its second and third generations. Throughout this work all pros and cons of each implementation approach are being deeply analyzed and solutions to the problems are explained. Therefore, the reader of the thesis will not only understand the technical aspect and many subtleties of the implementation, but will also see, why the user interface framework was initially built (and reimplemented later) one or another way.

Annotatsioon

Käesolev töö tutvustab kontseptuaalnset lähenemist kasutajaliidese raamistiku realiseerimiseks. Raamistiku uurimis- ja arendustöö oli teostatud autori poolt Customs Engine platvormi raames teadus-arendus ettevõtes Cybernetica AS. See töö hõlmab täieliku arenguprotsessi alates kõige esimesest raamistiku versioonist kuni selle tesie ja kolmanda põlvkonnani. Antud töö ulatuses iga realisatsiooni lähenemise kõik eelised ja puudused on sügavalt läbi analüüsitud ja probleemide lahendused on lahti seletatud. Kokkuvõteks, antud töö lugeja saab aru mitte ainult tehnilisest aspektist ja realisatsiooni peenust, vaid samuti näeb, miks kasutajaliidese raamistik oli esialgselt ehitatud (ja hiljem ümbertehtud) ühel või teisel viisil.

Table of Contents

Introduction

The business today is becoming increasingly demanding. The rapidly changing world significantly influences a wide variety of domains, forcing them to adapt. Any adaptation usually means applying a number of constraints to the business processes that should not necessary be beneficial for a particular organization.

Instantly developing information technologies have extreme power for solving this kind of problems. Automation and resource economy are the basic attributes that software solutions provide and are of the most important values for business. Successful software solutions indeed save a lot of time, making significant amounts of computation, analysis and information exchange that are no longer need to be performed by humans.

We may announce that today software plays the most important part in the modern world, having great technical potential, probably limited only by the human imagination. This may sound amazing, but does it mean, that having such a great potential, software development is a very complex process and takes time to build a product? Yes, that is true, the development process takes time, which is the most expensive resource business world often does not have.

When we speak about development time, we often mean a time schedule for a particular project. A year or two is a schedule, that may sound sensible to build a more or less complex system from the ground up for a mid- and, maybe, large-size business. However, it is hard to build systems of the same scope and quality, say, in several months. Organizational, analytical and technical processes of software development take their time and IT-people, should always try to optimize the development efficiency.

In my opinion, the technical aspect of software development process is the one, that requires special attention. This is because the technicians have relatively more freedom and space for speeding up the development and improving the product qualities. This is achievable by using a set of modern tools, applying architectural and design patterns the

right way or building components of the system in terms of some framework. Also, code reuse is usually a good practice and makes development easier.

This thesis is about building a framework - the framework, that is intended to make programmer's life easier, promote code reuse, allow high extensibility and good maintainability. The problem domain is user interface, but the implementation of widgets is not going to be discussed here. The goal is to describe the whole evolutionary process of the conceptual idea this user interface framework is based on, concerning key design decisions and technical analysis of the benefits and the trade-offs met. Because of the fact that given user interface framework is being developed within Customs Engine platform at Cybernetica AS, the next chapter will tell what the purpose of this platform is and briefly cover its components.

1The Customs Engine platform

Customs Engine (further CuE) is the platform that the author of this thesis and some other people at Cybernetica AS are instantly developing and improving. The reason, why CuE exists is very simple – it is code reuse. Because of the fact that at the moment a bunch of customs systems is being built by Cybernetica AS, it is sensible to identify common components that would share common functionality. This approach dramatically reduces maintainability issues and makes the overall development considerably easier. In the the following diagram the basic components of the CuE platform are presented.

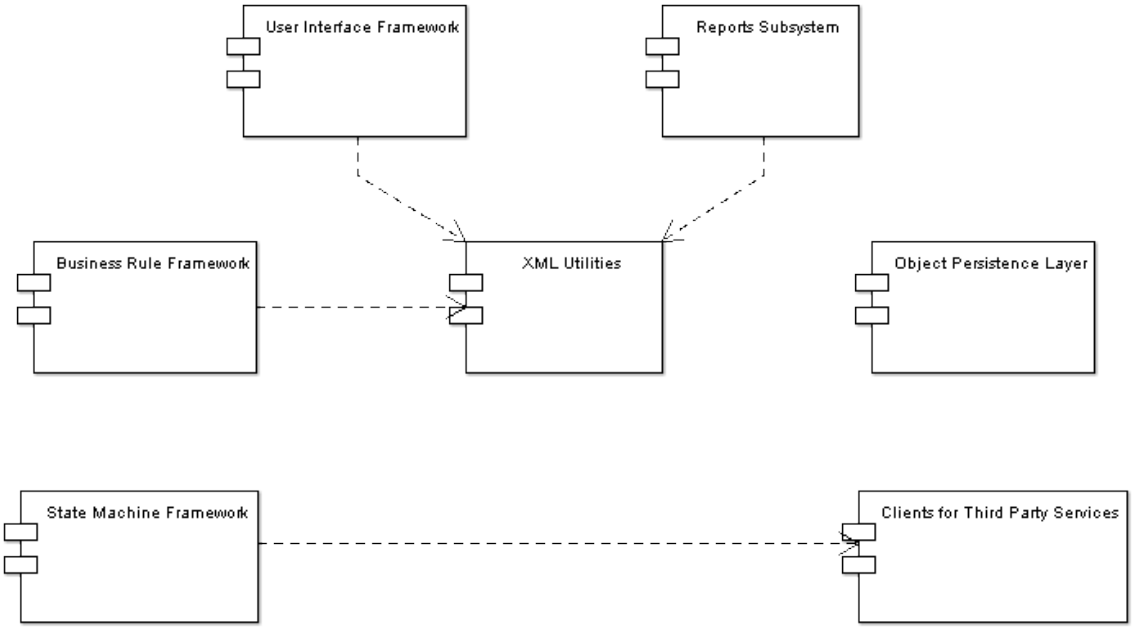

Figure 1: Basic components of CuE

As it can be noticed from this diagram, CuE platform consists of quite a few basic components, but at the same time, it covers the classical three-layer system architecture (that is data, application and presentation layers), also exploiting some additional components with useful functionality. Now lets examine each component in a little deeper level of detail:

- User interface framework is a layer, which is quite natural for almost any information system we may take today. This particular framework basically consists of a screen definition language (which was developed specifically for the given platform) and a number of widgets we might be used within this language.
- State machine framework is basically the engine for processing finite state machines

and is usually unseparable part in every system being developed within our organization (Cybernetica AS). State machine framework takes care of state transition mechanics, state validation (checks, if it is allowed to enter the state) and allows executing custom code if it is necessary to do some special processing.

- Object persistence layer is an abstraction level over Hibernate [1], which main purpose is generating object-relational mappings and building database queries, where any typos you make are always detected at compile time.
- Business rule framework allows writing and running validation rules on XML-documents [2]. Because there are often cases, where in some conditions value of one filed in the XML-document depends on the value of another field, this framework gives the opportunity to run such custom checks.
- The reports subsystem allows easily create SQL-reports. There is nothing complex about this component. Its general idea is running SQL-statements and displaying the results in a user friendly way. Because almost every customs system must have reporting capability, this component is a part of CuE platform.
- Heavy use of XML resulted in writing helper-code for XML processing and XML utilities component is the place, where such code lives. As XML is quite flexible and comfortable way of representing data, it is used by many components of CuE, which in turn use utility code from this particular component.
- The last component examined here encapsulates client code for third party services, which source code is not available. By means of this component, the customs systems are able to federate and exchange information with the third party systems.

Thus, the basic components of CuE platform have been covered. It should be now clear for the reader, what the platform consists of and what useful functionality it provides. However, the scope of this thesis narrows only to the single functional component and, essentially, the only one the author of this work developed the most – the user interface framework. The next chapter is going to describe the original concepts and design of the CuE user interface framework far before the problems of complexity were met.

2The original design of the CuE user interface framework

Actually, the initial implementation of the user interface framework of CuE was not written by the author of the thesis. However, further developments, refactoring and improvements were successfully implemented by him, which led to the second and the third generations of the framework. For better understanding, the newer ideas and approaches require some background from the reader, which is sensible to build on top of the very first – the original design of the user interface framework. This chapter is going to provide all necessary background and tries to create somewhat technical image of what the framework looked like way in the very beginning.

The user interface framework of CuE platform has two main concepts, that should be emphasized: the user interface definition language and XML-based forms. These two concepts together form a very powerful system for describing user interface structure and screen-to-screen navigation. Here we are going to look under the hood of these concepts and see the technical implementation with the design approaches used.

2.1User interface definition language

The main reasons for inventing the user interface definition language were the following:

- make building interfaces as intuitive as possible
- hide unnecessary technical details

Consider the first one. Having an intuitively understandable programming language, which does not imply any technical background from the person, who is writing in it, can considerably speed up the development and reduce the maintainability issues. Of course some basic syntactic rules have to be taken into the account. But these syntactic rules express problems in a way very similar to the way humans think. To make things clear, lets take a simple “Hello World” program in Java and Scheme [4] programming languages:

```
package com.foo.bar;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
```

```
        }
    }
```

Figure 2: Representing "Hello World" in Java programming language

```
(display "Hello World")
```

Figure 3: Representing "Hello World" in Scheme programming language

The program in Java language looks far less intuitive than the program in Scheme. When it is necessary to solve a problem of printing out a "Hello World", one may think that he or she must have a "Hello World" statement and display it onto the screen. Take a look at the code in Scheme – a problem is solved exactly the same way ordinary people think they should solve it. Whereas the implementation in Java is significantly more complex, contains a lot of technical details (which is necessary for successful program compilation and assembly) and does not directly reflect the problem.

When people (the programmers basically) think in terms of Java language, they do not usually think about the problem the way it was originally stated. The problem is being transformed to a bunch of technical details, which are combined and put together to form a solution. These details make programs far less intuitive and harder to understand. Thus, the idea behind the user interface definition language is to describe the problem and not concerning the technical details (where the problem domain is describing the navigation and the structure of user interface screens).

Here we smoothly came up to the second reason and the conceptual point of the language – hiding the technical details. But what details this user interface layer is intended to hide? The answer is simple – something, a programmer does not want to care about and this "something" is a HTML presentation code.

HTML-based user interfaces are extremely popular today, because HTML is very flexible technology. This also means that HTML code with all its power may become clumsy and bring all sorts of maintainability and portability problems. This is why this interface definition language completely hides HTML part, so that ordinary programmer only has to concentrate only on the problem. This makes sense especially when there is just no need to modify HTML. All customs systems that are being built on top of CuE platform have standard HTML interface and the commons style, so no single developer should care about look and feel. The question "how" does it looks and feels has been

answered once and the only thing that should be specified with interface definition language is what will be on the screen – the details of HTML rendering has been taken care of.

To make things more clear, in the following figure a small code snippet of the user interface definition language is introduced.

```
(screen orderList
    (title "Tellimus")
    (page-table OrderLine (update 'order.getLines()')
        (col "Kauba nimi" .productName)
        (col "Ühiku Hind" .price)
        (col "Kogus" .quantity)
        (col "Kogu hind" .totalPrice)
        (col (button removeLine "Kustuta"))
    )
    (buttons
        (submit refresh "Uuenda")
        (submit ok "Edasi")
        (submit cancel "Tagasi")
    )
    (event removeLine
        'order.removeLinesById(submitId);'
        (add ORDERLIST)
    )
    (event refresh
        (add ORDERLIST)
    )
    (event ok
        (add ORDERDETAILS)
    )
    (event cancel
        (go-back)
    )
)
```

Figure 4: Defining a screen in user interface definition language

The user interface framework of CuE promotes heavy use of this language, which is

because of its simplicity. When thinking about the problem of creating a screen, the presented code snippet above will look much closer to the way humans think than any possible implementation in some general-purpose language like Java. If a screen is needed, it is sufficient to provide the name for it. The same works with declaring titles for screens, where the text is the only required information that should be provided by developers. Normally this language requires as input only the information that have sensible meaning for humans, who has no special technical skills.

Lets now see, how to define a table. Everybody knows, that a usual table consists of columns and fields. Oh, probably the table consists of rows and columns, having cells in the intersection of the first two. This sounds quite logical and there are no arguments against it. However, in my opinion, it is not important, how people see things. The most important thing is what an object conceptually is. The concept of table is the same no matter physical paper or virtual software implementation of table is dealt with – it is the two dimensional way for organizing and representing data. It means, that when looking at the code above, it is *a priori* known, what a table is and, hence, even if someone does not know a specific syntactical element (*page-table* in this case), he or she simply makes assumptions, which are eventually correct. The same way anybody can guess what a mysterious *col* element does and what information should it be provided with. The concept of buttons is very common also. It is usually clear in advance that the purpose of a button is to perform an action or to spawn an event (lets put it this way). The point is that this user interface definition language provides means to define interface elements in a way that is conceptually close to the way humans see and understand things, at the same time hiding all unnecessary technical details which, in the given domain, are not of particular interest. These means usually allow building user interface screens of the CuE-powered systems in a matter of minutes.

There is one more critical feature of the user interface definition language we need to be aware of. Because the user interface definition language is implemented in Java, it allows to insert Java code almost at any place, in order to assist or customize the user interface rendering process. In fact, ordinary Java classes are generated from the user interface definition language, so there is always a chance to see the implementation code. But why would it be so important? Why is it necessary to emphasize the fact that user interface definition language generates pure Java? This is important, because this is the natural way of integrating the user interface definition language (and actually the

whole user interface framework of CuE) with the rest of the system that runs on top of CuE. This means that it is always possible to get control over what to render and when to render it – this is a good flexibility aspect and it is going to be discussed soon.

To sum up, the user interface definition language is a quite powerful and flexible way of telling the system, what is required to be on the screen. At the same time, this language sustains simplicity, so that not necessarily experienced programmers is the only people who understand the way things work. Thus, these properties of the language are very essential when building highly informative user interfaces with complex navigational logic.

2.2XML-based forms

XML-base forms are the second important concept of the CuE platform user interface framework. The idea behind XML-based forms is that user interface rendering may be based on the XML-document. Such approach makes development a lot easier, when you need to work with forms, where the data you display is read from XML [9]. It is quite hard and inconvenient to make such forms by hand exploiting user interface definition language discussed earlier.

However, there is certainly some information, which still has to be provided to the system and written by hand. First of all, it is necessary to give each field of XML document some unique name and write a mapping file. Second, describing the structure of the screen is needed. Describing a screen structure for your XML-form has to be done using names of the fields, defined in the mapping file. And final and the most important thing is writing a handler for the XML-form. In further chapters, all that is going to be discussed and explained in a slightly deeper level of detail.

2.2.1Making a mapping of XML-fields

The mapping of XML-fields is a very simple concept – enumerate the fields you want to obtain access to and give them names. In other words, a name is given to each XML-element, identified by unique XML-path (XPath). By doing this a compact overview of all the elements is achieved. Moreover, such overview promotes flexibility also. Consider a situation, where at the early development phases the XML-schema of the XML document is quite unstable. It means that programmers may often play with XML-elements, renaming them frequently or changing their location within the document. Having such single mapping file is very useful, as you always make changes

in one place. In the following figure a snippet of such XML-document mapping file is presented.

```
(values
     (80B 'Lehtede arv'
          (attr xpath "xmlVoucher/tirCarnet/vouchers"))
     (50A 'Kehtib kuni'
          (attr xpath "xmlVoucher/tirCarnet/validUntil"))
     (80K 'Lehe number'
          (attr xpath "xmlVoucher/voucherNumber"))
     (27 'Laadimiskoht'
          (attr xpath "xmlVoucher/loadingPlace"))
     (8B 'Saaja nimi'
          (attr xpath "xmlVoucher/consigneeTrader/name"))
     (2B 'Saatja nimi'
          (attr xpath "xmlVoucher/consignorTrader/name"))
     (50K 'Esitamise kuupäev'
          (attr xpath "xmlVoucher/declarationDate"))
     (50L 'Esitamise koht'
          (attr xpath "xmlVoucher/declarationPlace"))
     (80S 'Esindaja nimi'
          (attr xpath "xmlVoucher/representative/name"))
     (80AF 'Tõkendi ID'
          (attr xpath "xmlVoucher/seals/identity"))
     (80CL 'Paigaldaja'
          (attr xpath "xmlVoucher/seals/installer")))
```

Figure 5: A mapping document for XML-elements

The above example has been taken from TIR customs system, that has been built on top of CuE, featuring its user interface framework capabilities. In this snippet we see, that each XML-path maps to the name, which consists of two parts – the programmatic identifier and a title of the field. Take the first line, for example. In this case, the programmatic identifier is *80B*, which is necessary when defining a XML-form screen structure. The title is *'Lehtede arv'* and it is exactly what gets displayed over this field in the user interface. As you can see, this document is quite simple and carries no more informative contents except these presented above. However, the information encapsulated in this mapping file is a fundamental block for building much more interesting things, which are going to be discussed very soon.

2.2.2Defining a screen for XML-based form

The second step towards building a fully functional XML-based form is to define its screen structure. Defining a screen structure means distributing all XML-document

fields between the form tabs. Current implementation of user interface framework requires that you assign each field a single tab it should belong to. This makes the overall organization of widgets more convenient for the end user.

Another aspect of XML-based form is that by default an ordinary input widget is assumed behind each field. Additionally, if a simple input field is not enough, it is possible to write custom widgets of custom behavior. For example, the need for a special widget may occur, which would embrace several fields and contain a button for validating the content. In this case a separate widget class is being written and applied in the screen structure definition file. The following figure will illustrate the concept:

```
(tabs
     (common "Üldandmed")
     (goods "Kaubad"))

(values
     (80A common)
     (80B common)
     (50A common)
     ((50B 50C 50D 50E 50F 50G) common
            ComplexPartyCodeField "501B-G Valdaja")
     (50H common)
     (50I common)
     (80C goods)
     (80D goods)
     (80E goods)
     (80F goods)
     (80G goods)
     (31A goods TextAreaWidget))
```

Figure 6: An example of screen structure for XML-based form

The *tabs* syntactical element defines what tabs are going to be on the form. In this case, there are only two of them and their titles are "Üldandmed" and "Kaubad". The *values* element defines the mapping between names of the fields and what form they should belong to. Please note that two custom widgets are used here, TextAreaWidget and ComplexPartyCodeField, where the latter is a multi-field widget and groups several fields together.

This is basically all magic that should be taken care of when writing a fully functional XML-based form screen definitions. However, when digging deeper, it becomes clear, that something else is needed. Consider a scenario, where there are two different users that have different security roles. Each user should see the XML-based form of the same structure, containing the same dataset. But what if only one of them should be able to

modify the data? It means that there should be some additional concept which could assist in controlling the accessibility of widgets on form. Some generic mechanism is necessary, which could make widgets read-only, hidden or event filter out unnecessary elements and remove entire widgets from form. The user interface framework of CuE platform features such concept, which is called *profiles*.

Profiles are a tremendously flexible way for regulating accessibility of XML-based forms. Each profile defines, which fields will be accessible for modification, which must be hidden and which should be removed from XML-based form. Essentially, the profile is a mapping file, that maps each field name to its accessibility modifier. A usual profile looks something like this:

```
(set-fields
  (required "19" "31C" "44A")
  (read-only "1C" "7" "32" "81F")
  (hidden "80AE"))
```

Figure 7: A usual profile for XML-based form

The profile presented above means that fields 19, 31C, 44A are marked as required, 1C, 7, 32, 81F as read-only and a single field, 80AE will be hidden. Before a XML-based form will be rendered, this profile is run and its configuration is applied. As a result, the user must necessarily fill the fields 19, 31C, 44A, will not be able to modify fields 1C, 7, 32, 81F and will not even see 80AE.

As you see, XML-based forms expect screen structure to be defined and heavily relies on it. But there is still something important left – something that makes the concept of XML-based forms work. This is a concept of XML-handler and it is going to be covered in the following chapter.

2.2.3Writing a XML-handler

XML-handler is a third concept of XML-based forms technology that CuE platform provides. XML-handler is a simple Java class, which purpose is to assist in the processes that concern XML-based forms behavior (e.g. running special profiles or persisting a XML-document, which was modified by user). The main reason, why XML-handler concept exists is because there must be some way of customizing form behavior depending on the outer context or state of the environment (in other words, the means of configuration). So what is that mystical “outer context”? Well, this is something, that represents non-constant information, which may change at runtime and

what must be taken into consideration when working with XML-based forms. This basically means that if, for example, there are several profiles prepared, depending on some outer condition only one of them is run at form rendering time. The outer condition may be a button identifier a user had a chance to push. If a user pushes another button, a XML-handler detects that and appropriately adjusts the XML-based form behavior. Thus, XML-handlers are an important and inseparable configurable part of the XML-based forms technology.

Within the CuE project, the implementation of XML-handlers have varied over time, but the concept remained the same. The original implementations of XML-handlers were quite simple. They featured a specific environment variable (a plain integer number) as a mode identifier. Depending on the mode a XML-handler could select a specific behavior and preform specific actions on the underlying XML-document. Basically, there are several aspects that usually need to be customized:

- the profile to be run before the form rendering
- the title of the form
- the process of validating user changes of XML-document
- the process of persisting the underlying XML-document
- other call-back procedures, required by the framework

As was discussed above, a screen may have different profiles. XML-handlers are designed so, that they help choose the profile depending on the situation. In the original implementations of XML-handlers, the profiles were defined as constants, that could be chosen depending on the mode number. Consider the following code snippet:

```
private static final Object[][] PROFILES = {
      profile(PARTIAL_EXPORT, new PartialProfile()),
      profile(CONTROL_RESULTS, new ControlResultsProfile()),
      profile(REGISTER_EXIT, new ManualExitProfile())
};
```

Figure 8: The definition of profiles in XML-handlers (the original implementation).

The above illustrates, how profiles were defined in the original implementation of XML-handlers. This was a quite convenient way, because in order to get appropriate profile a mode integer number could be used as an array index.

The next, the title of the form, may also depend on the mode. However, the original implementation was very simple, as the method featured a standard conditional statements, which checked the mode and decided what title to return.

The process of validating user changes of XML-document is usually quite complex logic in XML-handlers. It involves different checks for consistency with XML-schema, running some business rules, that verify the content of the XML-elements and, sometimes, other specific manipulations with the document.

When dealing with document persistence it is often crucial to know, whether a completely new document is being created or an existing one is being changed. Therefore, it is necessary to perform some checks and query a database. Additionally, depending on the mode number it is possible to set some fields and store into the database some specific additional information.

There are many more other call-back methods that XML-based form technology requires a handler to implement. Among them are, methods that tell the framework when the document should be modifiable (this is in addition to profiles, but affects the whole document) or what screen description should be used in the current mode. It means that the initial implementation of XML-handlers relied heavily on this mode integer number and the logic within handlers depended on it. This may seem as not an elegant solution, but, nevertheless, it worked for some time and worked quite well.

2.2.4A brief technical overview of the concept

The following diagram introduces a brief overview of the XML-based forms concept. It illustrates the organization of basic classes, which are the fundamental building blocks of this technology.

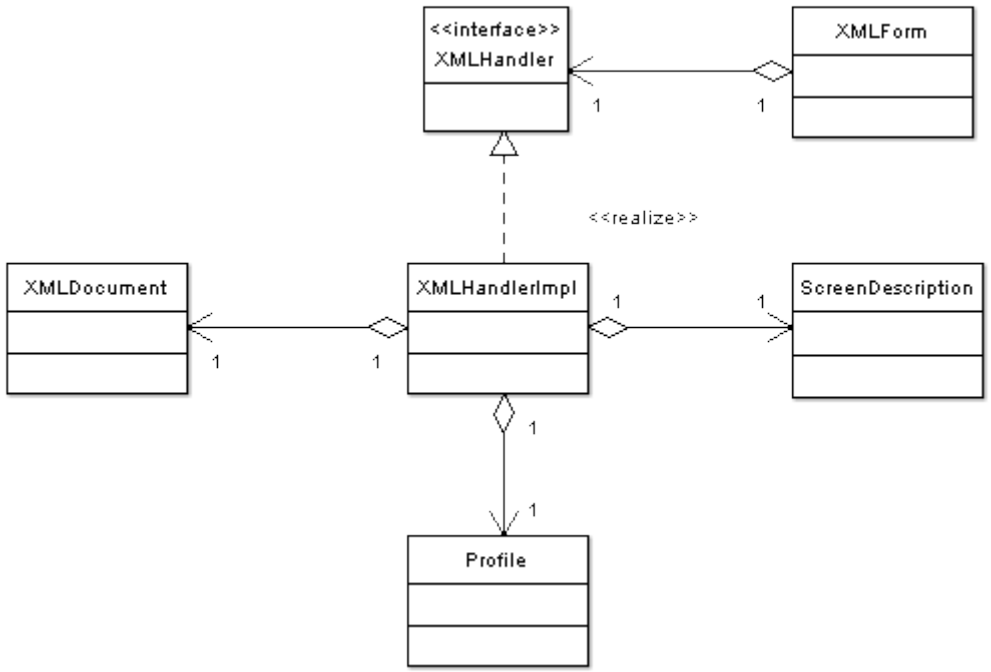

Figure 9: The basic classes used for implementing XML-based forms

XMLHandler interface and XMLForm class belong to the CuE platform. XMLHandlerImpl is a specific implementation of XML-handler, which is normally located in the module of some information system, built on top of CuE. Finally, ScreenDescription, XMLDocument and Profile objects have to be provided to XML-handler, which are used internally by XMLForm. Therefore, a basic XML-based form may be implemented with these several classes. But how this technology is connected with the rest of the user interface framework of CuE will be covered in the following chapter.

2.3Putting it all together

Now that, what the main components and concepts of the user interface framework of the CuE platform have been covered, it is time to see, how all these concepts work together.

Imagine a situation, where there is

a. an ordinary screen, written in a user interface definition language

b. a button on this screen that performs an action

c. a XML-based form that is being invoked by the button mentioned above

As it was discussed earlier, each button defined in user interface definition language has associated action (or event) with it. Upon each button click, this action is being invoked

and this allows to create a logical link between XML-based forms and screens made in user interface definition language. But what is the starting point of executing XML-based form? In other words, what method of what class must be invoked in order to give control to XML-based form?

The user interface framework provides such entry point, which must be given a XML-handler and a screen description objects of the XML-based form as input. This entry point is a static method of a specific framework class. It means, that there is no need to instantiate any object to execute that method and the only thing that needs to be done is binding XML-based form to ordinary screen and calling that entry point code from appropriate place. Eventually, the appropriate place for this code is no other than an event of a particular button.

However, there is still need to instantiate XML-handler and a screen description objects in order to make it all work. How can this be done? Is there enough information for instantiating XML-handler? Yes, there is. In the case of the screen description object it is extremely simple. The screen descriptions are defined in a specific language and at compile time the corresponding classes are being generated. We are totally free to instantiate the generated screen description class and use it were needed. Moreover, the generated screen description class already has a static field containing the instance of itself, so there is even no need to bother about instantiating anything. It is sufficient to just use the instance from that field. Instantiation of XML-handler, however, requires the mode number and XML-document to be provided. The mode number is usually given as a constant integer. This is because each button is responsible for a single mode, which basically selects a specific behavior for XML-based form. Now where does the XML-document may be found? This user interface framework does not apply any restrictions on where to get XML-document, so the most common source is the database. Usually, when a button is being clicked and an action is invoked, a database query is run, which fetches the XML that is immediately utilized by the XML-handler construction process.

To sum up, the core of the CuE platform user interface framework has been covered. The reader of this thesis should now be fully prepared for the following discussion of the problem. By now it might already be clear, that the framework itself has its limitations, which, when exceeded, lead to different problems of both technical and conceptual aspect. In the following chapter the problems of the framework are going to be considered and reasons, why they emerged are going to be discussed.

3The problem

The original implementation of the CuE platform user interface framework, which was discussed above, has been used for some time in several successful projects. However, when business demands became challenging and the complexity of systems being built on top of CuE platform increased, immediate measures should have been undertaken. Many problems of different aspects have arisen and needed to be solved as soon as possible.

The general problem was extremely simple, though. The user interface framework has met its limitations, what expressed in high code entropy. It means that the flow of logic became unclear and obscure for programmers and made any member of the development team apply much more effort in maintaining the code. In any way, it is very important to have clear and readable code as far as it is maintained by humans.

While the design and the overall organization of packages and modules still stood in place, huge and clumsy classes smelled as if it was just the right time for refactoring to come. However, the pure mechanical distribution of the large code into smaller pieces was not sufficient to make the entropy go away. It was necessary to change the way things worked conceptually, which meant changing the vision and understanding of the idea.

Thus, this chapter is going to discuss the problems that were met during the development of relatively complex systems using the user interface framework of CuE platform. At the same time, we will take a closer look at the root of each problem that has arisen, also concerning the scope of the changes that the solutions required. In the next sub-chapter we start discussing the first problem – the problem of readability.

3.1Readability

During the software development process it is quite usual for a programmer to face the problem of code readability. Same problem was met during the development of larger and more complex systems based on CuE platform, especially its user interface framework. Now lets take a little closer look at what exactly does the code readability mean.

The lifetime of code tend to be very complex. Often it involves all sorts of modifications and refactoring to implement necessary features. As a result some

readability problems may occur such as inconsistent variable, procedure or any other object naming and overall naming conventions, copy-pasted pieces of code, abstraction leaks, redundant complexity of algorithm implementations or (what is even worse) reimplementation of existing standard library routines. The usage of white spaces and overall code formatting style in general are also the properties of readability and may dramatically impede the understanding of code. Lets look at these properties separately and emphasize the ones we faced when building relatively complex systems based on the user interface framework of CuE platform.

The consistency of naming objects is essential in programming. Consistency means, that a name of an object reflects the purpose of the object. For example, a method should do what its name says it is intended to do. So, it is not quite intuitive, when you see a setter or a getter method, that does some additional magic besides just setting and retrieving a value. The case with variables is a little simpler and may say that a good name for a variable is a name that reflects its contents. However, today we quite often deal with object oriented programming, which means we have to consider classes also. But the naming of classes is a little more trickier than naming methods and variables. This is because a class encapsulates a number of methods and a number of fields, so that we should reflect the whole conceptual idea of the class when naming it. Sometimes this may be not a trivial task. Naming is important, as only the names can give you quick enough and brief enough overview of what is really going on inside some method, what is the point of variable or a class. Good names contribute to better understanding of code and, as a result, better readability.

When the user interface framework of CuE was being adapted to some more complex systems, some naming problems took place. Take as an example earlier discussed XML-handler that essentially is a way for XML-document behavior customization. As we needed additional functionality the XML-handler should had provided, additional methods had been added to the handler. Often, methods functionality did not correspond to the handler concept and they had to be refactored out to helper classes. Sometimes method names were not quite right and that made a higher level refactoring even worse. However, if we take a look at other properties of readability, all this was not that bad.

Copy pasted code is a quite common problem, especially in large projects. This usually occurs when a developer is unaware that the same piece of code has already been written by someone else. Even if you feel that with high probability the functionality

you are about to implement should already exist, the finding of necessary class or method may become a challenging task. All this leads to a copy-paste problem. But is it really a problem anyway? Well, consider a simple scenario of fixing a bug. Bugs occur quite often and may require a significant effort of finding and fixing it. When fixing a bug in copy-pasted code the effort increases multiple times, which also takes significant time. Avoiding copy-pasting of code is a good style as you are essentially not copying bugs.

Some copy-paste problems also occurred in the scope of CuE platform based user interfaces, though not often. Because of the fact, that the user interface framework of CuE is exploited in several systems, we sometimes observed that same logic was present in totally different systems. Also, because different developers work on different systems (even if the systems are based on the same platform), they are seldom aware of what is going on in the other system. Moreover, very few thought they were brave enough to refactor and extend the user interface framework. Although, it is quite hard to make such investments of time and strengths in extending the framework, they are obviously justified when done properly.

The next problem is abstraction leaks [6]. In general it means, that it is not enough only to be able to use abstractions - often, it is also important to be aware of how the abstraction layers are implemented. Consider the user interface definition language, screen descriptions and XML-field mappings, which are defined using abstract Lisp [5] alike syntax. However, they are not totally abstract, because in order to bind some of them together with XML handler or some other code in Java we need to know something about their implementation. Also, the user interface definition language provides us a way of inserting Java code practically everywhere we like. But combined with many lines of such Java code insertions the user interface definition language looses its elegance and becomes hardly readable. Take for example a situation, where we need to use different buttons depending on some specific condition, for example the state of the document. The original implementation of the user interface definition language did not allow to define buttons conditionally. It meant that we had to find some way of getting the buttons from plain Java method, where we had no restrictions imposed by the framework. As a result a helper class was created and tons of code concerning buttons was written. It was clear that very soon the user interface framework needed to be extended and appropriate measures applied. Despite the fact, such button

selectivity approach worked for a couple of projects, that was based on CuE platform.

The last properties of readability we concern in this chapter are redundant algorithm complexity and reimplementation of existing library routines. The algorithmic complexity may often be identified as a mess of nested cycles and conditionals having tens of lines of code. This is by no means the kind of code we would like to read (and probably to reverse engineer) every day. Actually, such code illustrates the mechanical thinking of a programmer, who did not think about the problem abstractly enough, at the same time totally ignoring means of abstraction [10] and decomposition the language provides. This leads to clumsy and unreadable chunks of code that are extremely hard to maintain and fix bugs. There were not so many examples of such code in the user interface framework of CuE, but there were some and they had to be dealt with.

The reimplementation of a library routines was an extremely seldom phenomenon, but we cannot pass it by. The reimplementation means that unaware developer writes the same logic, which already exists in the the standard library. Sometimes the reason for that is an algorithmic complexity, which hides the fact that some standard method does the same and, as a result, impedes reading and understanding the code as a whole.

Although, extending the user interface framework leaded to some complications described above, there were practically no problems with code formatting style and, in particular, white spaces. These are pure mechanical properties and integrated development environments do their job well. However, these properties affect code readability as much as the others described above and should never be underestimated.

3.2Extendability

During the development of larger systems based on the CuE platform one more significant problem emerged. The user interface framework of CuE began exceeding its limits. The problem of size and continuous growth of code had to be dealt with, which was the basic reason for the readability and maintainability problems. But what exactly grew so much that it was necessary to think about deep refactoring and, possibly, inventing a new concept of a user interface framework? Basically, there were three things:

- the screens described in user interface definition language
- the helper logic, encapsulated into a single helper class

- the XML-handler that was built upon the concept of modes

Take the first one. Because of the fact that it is extremely hard (or may be even impossible) to completely hide the details of the abstraction layer implementation, the user interface definition language in this case, we often had to write custom Java code within that layer. As the systems grew, their definitions of user interfaces also grew resulting into hundreds of lines of screen definition code plus additional supplementary Java code. Sometimes, due to the constraints imposed by the framework, different hacks took place, as the features implemented that way were not directly supported by means of the user interface definition language.

The problem of supplementary code has been solved to some degree though. The majority of the supplementary code was refactored out of the user interface definition layer and put into separate base classes. These base classes were extended by the interface screen definition files, what made the design of user interfaces a little more extensible and somewhat reduced maintainability headaches. However, the functionality hacks still stayed in place and they were the important signs, indicated that some concepts of the framework needed to be rebuilt soon.

The second problematic link was the helper class. It is exactly the same class we were talking about in the chapter above, which purpose was to construct button widgets selectively, depending on some external condition. Why was that a problem with larger and complex systems being built? Well, to some extent such approach was quite tolerable. However, the implementation of this approach was relatively complex, because in order to add a new button onto the screen, a programmer a) actually needs to add a button into the helper class and b) had to be aware of the things that are not of his interest, as helper class contained much more other helper functions, which were utilized both by user interface screen definition layer and XML-handler.

Consider situation a) – this is not quite good when building large and complex systems. As we know, abstraction layers assist us in handling complexity. In this case we deal with abstraction leak – the developer should be aware of the implementation. If we had several places in code, where such approach needed to be applied, the developer would have to know the specifics of the underlying button getter function or a specific helper class. Thus, this is certainly not the way we make the life of developers easier.

In situation b) the developer needs to deal with the code he or she is not supposed to care about. This is the major disadvantage of helper and utility classes, because their

scope is relatively wide. It also means that our helper class is of quite low cohesion because it lacks a single concrete purpose and responsibility. So, if the helper class is large enough, the programmer should deal with more unrelated code, what, in general, is not very convenient.

Finally, we came to the problem of XML-handler. As you already know, the original implementation of XML-handler had a concept of modes – a mode number came into the XML-handler from the outer environment and, according to the mode number, specific logic was run. The real problem here is that the mode switching was spread throughout entire XML-handler class and virtually every method had such selectivity logic. It was quite tolerable, when we had two or three possible modes XML-handler could be run with. But when systems required XML-handlers with more complex logic, more mode numbers, it was clear that existing approach could not work well. The basic reason was that the entire XML-handler grew very large and clumsy, at the same time making modifications and augmenting additional logic a painful process. This was due to the distributed logic for a single specific mode number throughout the multiple methods of the XML-handler class. For example, if you were to create a new mode number, you would have to create additional case into the method, responsible for the title of the form, as well as others like the method responsible for document persistence or document validation. Therefore, you may usually have some trouble when making this kind of modifications, because you have to make sure they do hot brake anything else. In the original approach of implementing XML-handlers there were no sufficient conceptual and technical isolation that could have secured the situation.

3.3The tight knot

Thus, we have denoted the basic problems of the user interface framework of CuE platform. All these issues were met while the development process and were the obvious signs of the need to refactor existing framework. The readability and extendability issues discussed above were a serious obstacles in a way of developers and needed to be solved soon. But what was the core of the problem – the thing that influenced the most on the decision to launch the process of rebuilding the framework? The condition of the XML-handler was the strongest argument here. Due to the reasons presented in the above chapter, it was almost totally unsuitable for further extensions and additional implementations of features. The helper classes also stimulated the decision, although they were still relatively usable and might stay extensible for some

time. Also, the fact that the helper classes and XML-handler were tightly coupled showed that in no doubt the current approach did not promote good design practices [3] and could be difficult to modify in future.

Some problems, concerning the user interface screen definitions also occurred. However, they were a little less critical, because they allowed quite high extendability even though some pieces of code could be implemented another and a more convenient way. At the same time, the user interface screen definitions exploited some methods of a helper class and XML-handler, consequently lacking a unified approach to the overall design.

Thus, the basic problems of the user interface framework have been identified and analyzed. In particular, interface screens, helper classes and XML-handler were coupled tight into single problem that someday necessarily needed to be solved. The following chapter is going to cover the solution that was worked out and what brought some light into the obscurity of the former design approach.

4Solving the problem

The above reasons were sufficient in order to start the augmenting and rebuilding the user interface framework of CuE. However, when we think more thoroughly, we see that the problems described above are not standalone problems – they are more the consequences of a design approach. This is the reason, why we are not going to do just the mechanical refactoring and decoupling of the helper and XML-handler classes in order to achieve some structural regularity or behavioral uniformity. If we do not think abstractly enough, with such modifications it would not be easy to achieve the necessary results, where the primary results are a) to create a conceptually different way of defining the XML-form behavior and b) to get rid of the extra helper functionality, making much of it regular and conforming to the new concept.

The first result we would like to achieve essentially means that we refactor the concept of modes in XML-handler. By this we would like to get something that would totally isolate the logic for each specific mode. Technically it means to get rid of the switch-case statements and to organize the logic into different classes. That would promote easier extendability and maintainability.

The second, extra helper functionality, should not be treated as such. In other words, extra helper functionality should become regular in terms of the new conceptual design approach. Of course, it is very hard to totally escape the helper functionality, but we must try to take it to the minimum. This could also contribute to easier and clearer organization of code and its further maintainability.

Now lets take a look at the problem from another angle and a little higher level of abstraction. If we try to conceptually unify the problems discussed above and the results we would like to achieve, we may see something really important. What we really need to achieve is very likely to be the design of homogeneous behavior and aggregating heterogeneous data.

Homogeneous behavior means that we need to treat the behavior of XML-handler uniformly throughout the whole user interface implementation. Due to code entropy and the limitations of the framework we sometimes had to deal with hacks, quick fixes or situations, where some way was the only way possible to implement a feature. It resulted in some amount of specific conditional situations that did not conform to the original design and were just plain ugly.

Heterogeneous data aggregation is essential, because we are going to deal with different aspects of XML-form: the title, the profiles to be applied, the properties of form validation and some more things we may probably need in the future. Basically heterogeneous data aggregation means the data, that came from different sources. It is easy to understand if we think that in one case we could need the XML-document data form a database, profiles from their definition list and the title as a constant. All this data has different source it comes from - that is why we chose to call it as heterogeneous.

All in all, in order to make things really convenient for developers, all that could be maintainable from the single place. This means, we could have something where to put behavioral and declarative (the data) logic together. Such approach isolates the XML-form logic definitions, gives the developer more confidence when making changes, at the same time freeing him or her from reading and worry about the condition of the unrelated code. These were the things we were trying to achieve during the process of rebuilding and augmenting the user interface framework and in the next sub-chapter we will see in detail what all that nice theory turned out to be in practice.

4.1The concept of scenarios and steps

In this chapter we are going to discuss the first version of implemented solution - the concept of scenarios and steps. Here we explain the reason why we decided to use this name, at the same time concerning technical aspect of implementation, defining the advantages we get and the trade-offs we had to deal with.

4.1.1The idea

Lets start from the general idea of the concept. Having seen the problem from different angles we noticed that each separate piece of XML-handler logic, previously identified by mode number, would be sensible to give such name as scenario. Scenario as such is a sequence of activities that may take place during the processing of a XML-based form. It usually starts from invoking some actions before displaying the form, then displaying the form itself, again running some supplementary logic and viewing some other form. As you see, we get here a kind of a cycle. However, we are not sufficiently precise bringing such an example here. You may have already noticed, that between viewing the forms we have to brake to process in order to give a user a chance to preform necessary activities. The question is, how are we going to do that?

To answer the above question we have to specify what the building blocks of our scenario concept are. Eventually, they are steps. Because scenario is usually a sequence of activities, running a scenario is a step-by-step process. This fact gave such name and we going to stick to it.

Now how are we going to break the sequential process of running a scenario and give the user a chance to make his own work done using the form on the screen? We may possibly solve this by assuming, that each scenario step is a finished process that begins with running some custom logic and ends with viewing the form on the screen. We should also mention, that such approach requires a scenario step to be a continuous and unbreakable process. However what does it technically mean? Imagine that we have a scenario that consists of several steps, what is basically intended to display several forms. If we brake the process after displaying the first form, it means to continue the scenario we have to start with the second step, not the initial one. It becomes clear that scenarios must store information of what step should be executed the next time the scenario is continued. In other words scenarios should have state.

The next problem is, how should we handle the state of each scenario? What the solution would be look like technically? Well, we will try to be as simple as possible and to work out a simplest solution for that.

Lets start from something we already know. We know that a scenario is a sequence of steps and we derived that a scenario should have a state. The state is intended to assist us in finding the right step that should be executed next, when earlier broken scenario processing continues. It may seem, that we may have to keep track of what steps have been processed and what have not been yet. This approach is simple enough and might eventually work an appropriate way. But still there is a simpler way. According to what we discussed above, our goal is to be able to aggregate all necessary steps into a scenario. It means that scenario *a priori* knows what steps are going to be processed. So, we do not need to keep track of steps – we do not have to store them in some container, because we already have them *a priori* stored. This leads us to an extremely simple solution of storing and the meaning of the state of scenario. The state might be represented by an index of a container, where all the steps are aggregated. Each scenario execution would increment that index by one, so that the following executions would use the step at the current index position. Technically each scenario might contain a collection or an array with aggregated steps, what may be indexed. This is the way, how

steps were organized and processed within scenario during the refactoring of the user interface framework. But would such approach be sufficient?

In practice, we also needed some kind of steps of a little lower level of abstraction – those that could run arbitrary code and would not require to finish with displaying a form. This required separating levels of steps abstraction, as we needed to run steps from within other steps – the composite ones. What all this really means is that composite steps should aggregate ordinary steps, which in their turn contain decoupled functionality. So that composite steps are only for aggregation purposes and the composite steps are the ones that are to be put into a indexed scenario container. Therefore, only composite steps should break scenario into parts instead of just an ordinary steps. At the same time, within a composite step, ordinary steps always run as a continuous and uninterruptible sequence.

Having talked about steps a little, its time to designate, what they essentially are. Steps as such have to be defined in a separate classes, but that does not mean they are conceptually classes. Although, in some conditions steps may have state, the general idea is to treat steps as actions or functions. This is because the single purpose of step is to be run and not accumulating the state. Steps should run the functionality coded into the specific method and this is the primary purpose for their existence. Each step implements a common interface that allows to treat and use steps uniformly. We can observe this uniformity if we take both composite and ordinary steps. As both of them are steps, they implement the same interface.- the only difference is the type of code they run. Composite steps sequentially run all other steps aggregated into them, whereas aggregated steps run business logic itself. Using the same interface scenario interacts with its aggregated composite steps. So, as we now see steps are built with a functional concept in mind and in such way allow decomposing the logic of scenario into multiple smaller pieces.

Until now we discussed the behavioral side of scenarios. However, the idea behind scenarios was not just to provide convenient means for configuring behavior (by combining steps), but also to make the declaration and configuration of the initial environment easy. Now it is time to take a look at a declarative side, the environmental data, that scenarios may aggregate.

What is the initial environment that we would like to aggregate in scenarios? The initial environment is basically a set of objects that are going to be used by the steps

aggregated in the same scenario. Among these objects may for example be:

- the title of the XML-form
- the XML-document
- list of profiles to be applied
- list of business rules to be run, when submitting the form into the database
- roles of the profile, which show whether or not the logged in user is authorized to run current scenario.
- different flags that may be used to control the behavior of XML-handler or individual steps

Such environment was essentially the rebuilt concept of modes of the original implementation approach of XML-handler. Instead of writing a bunch of switch-case statements in every method of a handler we set up the environment, what XML-handler uses. This means that we did not need modes at all – the handler became configurable in a much more object oriented way.

Now lets see, what does that environment look like technically? Well, in technical sense, the environment represents a contextual configuration or a context for short. Each scenario has its own context with specific initial configuration. The context itself is implemented using the JavaBean [7] convention. Essentially it means that we have a plain java object with fields of object state and accessor and mutator methods. The fields of object state contain the contextual configuration, whereas accessor and mutator methods only provide the way of retrieving and modifying those fields, which conform to *getSomeFieldOfState()* and *setSomeFieldOfState()* naming pattern. Besides this, there is no more additional functionality in contexts, so that they act more like data transfer objects [8].

However, context is not the way to only provide the initial environmental configuration to XML-handler and steps of scenarios. The second important role of context is to allow steps to exchange information by means of context.

Usually steps just read the configuration provided by context and run some logic depending on this configuration. But sometimes it is useful to exchange information through context. Some step may change an object that resides at the moment in the context and the steps that are run later can use this modified object. Such object may for

example be a XML-document that is usually present in the context. Quite often certain steps must generate some values and fill appropriate fields of a XML-document with them. At a later stage some step may check, whether a document had those fields set and, if necessary, run specific code. Moreover, some steps also may set necessary flags in a context, which are checked by the steps run later.

As you see, the context may be used to exchange information between steps. But please note, that such exchange is always a one way process. This means, that one step cannot pass some information, that would somehow be available to another step that is positioned in the execution sequence before. This just cannot happen, because there is only one pass through the execution sequence and each step is always run no more than once. By the way, it is also very hard to imagine why would we want to have this kind of behavior.

By now we already know what the context technically is. But we did not mention, how do we pass the context to XML-handler and steps. Further we are going explain this aspect too.

We discussed that a context is defined in terms of scenario. In other words, we write some context initialization code in scenario class, which uses earlier created context object. But where the context is being held in general? And in what way we get access to the current context?

To answer these questions I would like to remind that our user interface framework is web-based and that it is possible to obtain access to such information as web-request, server response and session. Web-request is essentially a request from the client, who would like to view our web-page. The server response is the system's response to the client request. The response is usually a web-page, rendered by the user interface framework. The session is a logical communication channel that is usually established when you log into the site and closed when you log out. In our case, a session is crucial, because of some of its properties. The key property of session we should pay attention to is the ability to store information between page hops. In other words, session allows to store globals in some sense. This property has been exploited with scenarios.

When scenario is being constructed, the context and step configuration are stored into the specific scenario fields. After, the scenario registers itself with a global session variable, so that it could be accessed later on. It means that the context may be found only by asking it from the current scenario. So that if in some place we would like get

some object from the context, we are supposed to be aware of the current scenario, the get context and get an object in the end.

Practically very often steps need to get a lot of objects from the current scenario context. The way of getting objects from the current scenario context presented above may seem complicated and not that convenient. That approach may seem a lot more inappropriate if we recall that steps as well as the context are created and configured in terms of scenario construction process. Having things work that way we get an outstanding opportunity to pass a context reference to all the steps we define in scenario. This reduces the chaining of methods we would normally have to do in order to get to the context and the required object. Thus, the automatic context registration mechanism has been implemented, so that each step of the scenario is aware of and has a direct access to the current context.

When talking about automatic context registration, some sort of abstraction is meant by that. The abstraction that has some implementation details, but which in general has little influence on the concept of scenarios and steps as such. We also have similar abstractions that hide implementations of running composite and individual steps, context construction process and registering it with current scenario. All these and many anther implementation details are intentionally hidden from and are shared between the concrete scenario definitions. In fact, there are several levels of such abstractions. Because of the fact that the current user interface framework is used throughout several systems the lowest level abstracts and provides for shared usage their the common functionality. This is the platform level of the framework. The second level encapsulates and shares the functionality between the concrete scenarios within a specific system. This may be called as an application level of the framework. Finally, the highest level are the concrete scenario definitions themselves. These use much of the abstracted functionality promoting relatively efficient code reuse.

Also, XML-handler has been refactored in a very similar way. The common system functionality has been analyzed and refactored out into the separate base class that belongs to the platform level of the framework. At the same time, this base class at the platform level is extended by the application level of XML-handler classes, providing the support for scenarios and contexts.

Helper class refactoring has also taken place. If you remember, helper class contained a specific method for getting buttons for the interfaces that are described in the user

interface definition language. This method has been refactored and, in particular, the implementations of buttons. Earlier, upon the invocation the button created a XML-handler, giving it a mode as a construction parameter. After refactoring, upon invocation of the button a concrete scenario is being constructed and its execution process is being launched. Thus, these buttons are essentially entry points to concrete scenarios and the helper class provides us the convenient way of retrieving them to the user interface definition layer.

Thus, the idea of scenarios and steps has been presented, the basic concepts discussed and some implementation details revealed. This concept may seem as a quite nice and suitable solution for the problem stated. But as anything in the world this solution has its own pros and cons. Below, we are going to cover some advantages and trade-off of the current approach for scenarios and steps implementation..

4.1.2Advantages

Lets start from the major advantage that is clearly seen in this new concept of scenarios and steps. As was mentioned above, any scenario consists of sequence of steps, both composite and individual ones. Also, steps has functional essence – that is they are just the pieces of code needed to be run at appropriate time at appropriate place. Steps can implement arbitrary auxiliary functionality, but basically manipulate objects of the current scenario context. What makes steps really special is the flexibility they allow. And the flexibility is achieved in a very simple way. If for example we would like to add some specific piece of code to the execution sequence of steps, we first have to write our custom step with necessary functionality and register that step in the execution sequence. That is all we have to do to set the custom logic up and running.

The second important advantage is also related to steps. Having such a flexible way of creating and using custom steps promotes easy code reuse. Steps allow to decompose the logic in such a way, so that it can be shared between different scenarios or even between scenarios of several completely different systems, built upon the common CuE platform. This fact dramatically reduces the lines of code developers have to write and more promotes taking a component (a step in this case) from the ”shelf” and use it. Thus, writing a scenario tends to be a process more about configuration rather than plain programming.

In general, the declarative approach of describing scenarios may be quite a strong

advantage. Having such approach it does not really matter (and the developer does not bother) what the implementation of the declared steps is, unless, of course, something special is required. Context is declarative as well, as you essentially initialize it with the objects that are very often constants. Thus, such declarative paradigm and the approach to abstractions may seem to be almost a panacea that would solve the problems of the former, original design. However, nothing gets for free and even such a nice concept of scenarios and steps has its own downsides. The following chapter will cover the trade-offs that were met when applying this concept in real development.

4.1.3The trade-offs

After the user interface framework has been refactored and started being used in the development of customs systems, some drawbacks have been noticed.

The first drawback noticed was the large amount of scenarios that had to be defined. Because of the fact that the developed systems had far non-trivial business logic, many different scenarios had to be created and put in use. From the developer's point of view it meant that the naming became a challenging task. The scenario naming is important here, because nobody normally wants to open each class and scan through several scenario implementations in order to find the necessary scenario and work with it. To prevent this some naming had to be worked out. But nevertheless, the names given were relatively long and it was not convenient to manipulate them in code. Although, names were descriptive enough to save a developer's time.

As the number of scenarios grew, there were some attempts to promote scenario reuse. Scenario reuse meant that when there are several scenarios with very similar logic, where some specific cases depend on some contextual flag or an external environment state, a single generic scenario has been created. In this generic scenario there were normally no default constructor, but the constructor (or several constructors) with parameters. Such parametrized scenarios were often used in many different places in code, emphasizing reusability. But what trade-offs this approach might have if it gives a nice code reuse capabilities?

The problem is in uniformity. Scenarios would be easier to deal with if there were a single uniform contract on creating and using them. The part of such contract is dictated partially by the extended classes of the underlying levels of abstraction. However, the underlying abstraction levels cannot dictate, how the subclasses are to be constructed.

The uniformity problem rises, when it is necessary to keep track of XML-document drafts. In the customs systems that were developed on top of this framework very often we had to deal with scenarios that modify the document and temporary draft was usually stored into the database. When the user wants to resume editing the draft, a database should also contain the information about the scenario, which would later be reinitialized and launched. Because the majority of scenarios had the default constructor, there was no problem with constructing it. With generic scenarios, which have non-default constructors, it is very hard to know how to build them, what makes using them with XML-document drafts far more inconvenient and complex. This non-uniformity illustrates, that there are cases, when we can use one type of scenarios, whereas there are also cases, where only another type of scenarios (generic scenarios) suit. Therefore, logically there are two types of scenarios, even when technically all of them seemed to be unified.

One more trade-off should be mentioned here. This trade-off is a global session variable that contains the scenario being currently processed. This is not a very good solution to use global session variable, because, as practice shows, some weird bugs may occur because of that. Such bug were related to the context and, namely, the XML-document object. The reason for these bugs was the fact, that sometimes XML-document object was created separately, not within a scenario execution process. This was real problem right after the framework has been refactored. Also, global session variables are not reliable enough. This is because they are shared and any code that uses it (steps for example) may break it, causing much more insane bugs than in the example above. Therefore, due to reasons above, global session variable for storing scenario information is considered as a significant trade-off and is the aspect to work on.

To sum up, lets take a look at the full picture of the rebuilt concept. The following class diagram (Figure 10) introduces the general design of scenarios and steps approach, which was widely applied in several successful systems, developed on top of CuE platform.

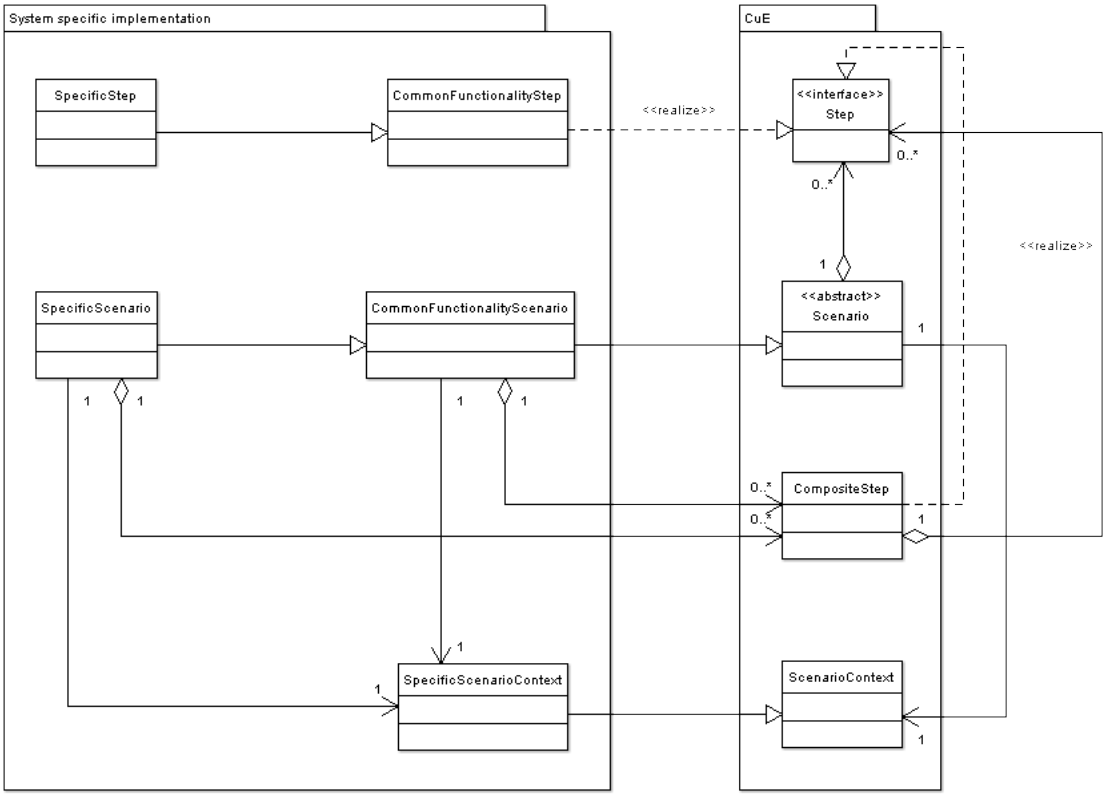

Figure 10: The second generation of the framework

The above diagram depicts the general organization of classes in the second generation of the user interface framework. As you can see, the model is quite complex and contains many relations between classes. However, despite such relative complexity, this design gave a conceptually different view of the problem, what made the development notably more efficient.

All in all, the main trade-offs of the first rebuilt version of the user interface framework have been covered. However, it is clear that the new refactored framework has its own problems too, although not so significant as the original implementation had. The following chapter will tell, in what way this version of the framework was improved, what concepts were applied and what difficulties were dealt with.

4.2Further refactoring

Due to some trade-offs described above a further refactoring should have been initiated. To some extent, the above mentioned approach suited the needs of the customs systems built on top of CuE. But again, as the systems evolved, some complexities were met and a stronger conceptual reorganization and unification was needed in the nearest future. If we take a look at the previously described concept, it may seem relatively complex. For

example, the conceptual separation of composite and individual steps – was that really necessary? Well, at some point this made the development of user interfaces easier in current domain. The developers changed their way of thinking about what user interfaces essentially consist of and looked at the problem under slightly different angle. But nevertheless, it saved some development effort and the general feedback received was positive.

At the same time, the created framework was not quite that, what it was meant to be. It was very close and almost that concept everybody in the team were looking for. Some structural complexity and non-uniformity was present and created some inconveniences. Therefore, the work on the new implementation began and its purpose was the improved concept that would promote uniformity, more conceptual simplicity and code reuse.

4.2.1The new implementation

The new implementation of the user interface framework pursued the following objectives:

- avoid global session variables
- build a functional concept
- achieve uniformity, where everything is a scenario
- buttons should be an inseparable parts of a scenario
- improve scenario initialization and construction processes

These and some other objectives are going to be discussed here. At the same time, the implementation details will also be concerned and explained where necessary.

4.2.1.1Avoid global session variables

Lets start from the problem of global session variables. A global session variable for storing current executing scenario was extremely important in the former implementation of the user interface framework. This is because it allowed a single access point, which was a quite convenient approach. At the same time, there were some drawbacks, that were presented above, which made the implementation of the concept less reliable and more error-prone.

The basis of avoiding global session variables was built on top of context passing

functionality. The context is the basic data carrier and according to the new idea was meant to be copied each time the next processing step was executed. It means, that the context was no longer shared between steps. Instead, each step had its own individual copy of the context and did not depend on anything global. This solution isolated the environments of steps and made developers feel more confident while writing scenarios, because they did not have to affect the global environment. Also, this could make developers feel that, because of such isolation their code in their scenario can hardly break anything. However, this assumption is not right. The reason is that the steps, which are run later can be affected, because they inherit the state of the context, possibly modified by the previous step.

All in all, a natural question may rise here – how, nevertheless, all of that is related to the problem of locating scenarios in a global session variable? The relation is relatively simple. Because of the fact that very often a context was a single reason why a scenario was accessed, it was necessary to change the way of passing context around.

However, changing the way of passing a context was not sufficient in order to get rid of the global session variable. It was also very important to change the way we pass the current state of the scenario.

4.2.1.2A stateless functional concept

The other reason, why a scenario needed to be accessed is to make it execute a following step. Of course, there are no arguments about that – steps are aggregated into scenario and, therefore, to execute the proper step, we have to know what scenario we deal with and, in particular, what is the state of the current scenario.

A scenario state is something that should survive multiple web-requests. It would be natural to use a session variable and store the state there. But is there other way of storing or passing a state? Well, in the new implementation of the user interface framework a stateless concept was invented. The idea of the concept is relatively simple, though quite elegant: each screen that is being rendered by the user interface framework contains buttons to perform some actions, where each button already knows some information about the step it should run if being invoked. Upon request, many different widgets (including buttons) are being rendered and at exactly this moment a necessary information about the next processing step is bound to the buttons on the screen. This allows to avoid storing the state, as each button already knows its own

piece of code it has to execute. Therefore, there is no more need to locate and determine the current scenario and, thus, it is possible to avoid using scenario state at all.

However, there was one more reason for storing scenario in a global session variable. Sometimes it was necessary to find out, what scenario class is being dealt with. It usually found its use with XML-document drafts mentioned in one of the above chapters. In order to store a document draft, it was necessary to know, what scenario was running. In order to properly restore the environment later, a scenario class name had to be stored into the database, together with the XML-document draft. Then, the class name had to be read from the current scenario, which was located in the global session variable. In order to avoid such global variable at all, it should be possible to work out another way of determining a class of a scenario. In the new implementation, a scenario class name was stored into the context, which was quite an obvious solution for that. Later, when the class name is required for storage with XML-document draft, it is supposed to just be taken from the context.

Therefore, it was totally possible to develop a stateless concept for the new implementation of the user interface framework. Also, having no state gives the concept a different look and more reliable functional development approach.

4.2.1.3Laziness

At this point functional properties of the new implementation approach are clearly seen, where the basic property of concept is the absence of scenario state. However, there is one more important property that should be considered here – the laziness.

The concept of lazy objects is an extremely important and is well known as a separate design pattern. Some programming languages also directly support this pattern. Scheme has a concept of *promises* which may be *forced* and executed later. Smalltalk has *blocks* that are a part of a language and are the technical means for concept of lazy object. Despite the fact that there are different names, the idea is the same.

So why is the idea of lazy objects so popular and why do developers often find it useful? Lazy approach is sometimes necessary, when large object graphs are concerned. In other words, when the object structure is relatively big, it is usually not a very good idea to hold everything in memory. It is more appropriate to load only these objects what are needed at any particular time. Of course, this approach does not suit to some specific intense object tree walking algorithms and this is actually beyond the scope of

the problem (as the new implementation of the user interface framework does not do such intense operations and does not parse any object trees). But nevertheless, the pattern of lazy objects was vastly applied throughout the design.

If you look at the stateless approach at a little different angle, it will become clear how the laziness concept fits the design. The general idea was to make the rendering of the user interface screens as lightweight and as fast as possible. It meant, that at rendering time each button had to be bound with minimal amount of information, so that it would also be able to run appropriate code based on that bound information. Now the most important aspect here is that only the binding is relevant – no object constructions or other code executions could take place at page rendering time. In other words, at page rendering time each button should be only aware of what is going to be run when it is clicked, whereas the construction and execution of this "what" should always be initiated within the button invocation process. This approach avoids creating tons of useless objects at page rendering time, thus making this process relatively lightweight. Moreover, there is really no need for these objects to be created before the button was clicked. It would be waste of memory, if for each button significant amount of initialization code had to be run building many objects that may possibly be used. But user does not necessarily needs to click all the buttons within his web-session, so much of previous initializations and preparations would be just useless. Therefore, it is obvious that lazy approach of the new implementation of the user interface framework brings efficiency and significant memory savings.

4.2.1.4Everything is a scenario

Up to this moment it was told, that each button of the user interface screen is aware of some piece of code that is to be run upon each button click. Also, in one of the above chapters it was mentioned that the code button runs is conceptually a step of the scenario. However, when the stateless scenario concept was invented in the new implementation of the framework, the attempt for scenario unification and generalization was made.

The scenario unification and generalization meant that there would be no more scenario steps (neither composite nor individual ones). Instead, scenarios were designed so that they would aggregate other scenarios, which would execute code just like steps in the former design approach. Such concept is a potentially strong means of abstraction. This allows to build three and more abstraction layers of scenarios, what could allow better

and finer code reuse, though having some overhead because of large number of scenario classes developers would need to define. Basically it means, that there are two logical types of scenarios – aggregate-scenarios and the lowest level scenarios, that actually encapsulate and run custom code. But thanks to uniformity achieved, these two scenario types are handled the same way and technically there is no difference in implementing them.

4.2.1.5The implementation details

As the technical aspect has been concerned, it would be useful to describe, how the new implementation of the framework is built.

The new implementation consists basically of two types of objects - scenario and a context and no composite or individual steps exist in this new approach. Each scenario basically consists of three methods:

- a method for context initialization
- a method for button definitions
- a method that runs custom code

A method for context initialization already has available context variable. The code in this method should appropriately initialize the provided context and set up all the environment that will be available to the code later. A context object itself is an ordinary map. But having a map, a set of keys is necessary. Moreover, context keys should be consistent throughout all scenario implementations, so that it would be possible to read some object from the context at a later stage. To protect developers from typos, a common interface was created, where all keys for the contextual environment are defined. Each scenario simply implements that interface and all constants become available straight away. Using these inherited constants, any developer can easily write context initialization code and be sure no typos and inconsistencies ever occur.

But was that the best way of implementing and working with the scenario context? Alternative could be to implement a simple bean or data transfer object for storing contextual information. This approach could give static type control as an advantageous property, so that developers would not need to do an insecure type-casting operations. On the other hand, in rapidly developing system a contextual structure is more likely to change, what makes it quite inconvenient to modify the context class. In this case it is

required to add a field, write a getter and a setter. Compared to the ordinary map it is more work to do, because with map it is only sufficient to define a constant for a key. Thus, this may seem a kind of philosophical question about what is better here. But at the time of inventing and building the new implementation of the framework, map-based context was preferred due to easier modifiability.

Now lets see, what is a method for button definitions. As was mentioned above, a button is essentially an entry point to scenario. At the same time, arbitrary piece of code can be bound to a button. Putting this two things together it is obvious that a button can also execute a scenario. Because of the fact, that in the new implementation a scenario contains from other scenarios, defining buttons within one scenario is now a natural way to aggregate other scenarios. This approach assumes, that behind each button is a code, which, when the button is clicked, constructs a specific scenario and executes it. Therefore, within a single primary scenario it is possible to define buttons, which are entry points to other secondary scenarios.

Within a method for button definitions a developer basically constructs a button object and appends it to list. The construction of a button is intended to be a transparent process, which requires a developer to only execute appropriate method, given it a title of a button and a scenario to be bound. It is also possible to construct buttons depending on some condition, where the condition may be controlled by a context variable. At buttons initialization time the initialized context is already available, so that sometimes it may be of great use. All in all, there are no complicated activities required from a developer, so the button definitions may be considered as an easy and a quite straightforward procedure.

Finally, a scenario method, which is intended to run custom code should be concerned. This method is also usually written by a developer and is one of the most flexible parts of scenarios. In this method many sorts of programming logic may be. There may be some auxiliary code for making different pre-processing actions, at the same time some secondary scenario may run or a user interface screen rendering process may be triggered. Also, initialized context of the current scenario is available, which allows additional flexibility of code. In general, this method actually encapsulates something, what scenario should do when run, so that all business logic must belong here. Moreover, the general purpose for scenario existence and the final step in its life cycle is running this method.

To sum up, each scenario of the new framework requires three methods to be written in order to operate properly. Also, there is additional supplementary functionality, which makes button definitions extremely easy. This supplementary functionality is organized into the class, which represents lower level of abstraction and is extended by scenarios. Thus, this abstract class makes all convenient helper methods always available from within any scenario, what makes scenario-based development significantly easier.

4.2.1.6Binding with the rest of the framework

In the previous chapter initialization of scenario context and definition of buttons was described. The concept is quite clear, how in the general case the context and the buttons are intended to be used. However, the technical part is still not quite clear. How do the context and the buttons are being exposed to the rest of the framework – that is to the user interface screens written in specific definition language, mentioned in the very beginning of the thesis? To answer this question, it is necessary to concern the mechanism of scenario invocation process.

In order to invoke a scenario, it is necessary to call a specific *go()* method. This method belongs to an abstract class, which is normally extended by other scenarios. Actually, the *go()* method is a common interface for all scenarios of the new framework implementation and performs the following basic operations:

- launching context initialization code
- launching button definition code
- registering context with web-request
- registering buttons with web-request
- launching actual scenario business logic code

Launching context initialization code means that code written by developer of some specific scenario is run, which results in proper set-up of the context data. The same concept applies for launching button definition code, where buttons are being constructed and aggregated into the list. Registering context and buttons with web-request is the most important moment here, because web-request is essentially a link with the rest of the framework. Now having registered the context and the buttons as the request attributes, it is possible to get them at a later stage. It means, that when scenario runs user interface screen rendering code (where screen was previously written

in the user interface definition language) it is possible to interfere into this process. In order this to work, the user interface screen must have some modifications also. Because of the fact that the user interface definition language allows insertion of arbitrary code at almost arbitrary point, this makes it possible at rendering time to get from the web-request previously stored context and buttons, which may be used to customize the rendering process. Context may be retrieved and some specific widget rendering behavior may be possible. However, buttons are the crucial part here. From the web-request it is possible to get the buttons defined within a scenario, which, in their turn, are then capable of invoking other secondary scenarios or, again, some arbitrary functionality. Therefore, with this new conceptual approach great flexibility is achieved and enormous opportunities of the framework emerge.

4.2.1.7Still call it "scenario"?

Having discussed the new conceptual approach of the user interface framework, a natural question may rise: do scenario classes of this new implementation actually satisfy the definition of "scenario"? In fact, this is a very difficult question. Indeed this current approach to the problem differs from the former one. In general, a good definition for the user interface scenario could be something like a set of screens a user may navigate, where the order of navigation may be unimportant. The common contract of the new scenario concept is that each scenario class execution may run any code (including running other scenarios) and view some screen. Whereas the former implementation dictated, that a scenario is a set of steps that are run sequentially. Therefore, it means that the current new implementation could have different and more suitable name, because the overall approach is more close to be based on steps rather than scenarios.

4.2.2Benefits achieved

Knowing, how the new implementation work, it is time to analyze, what benefits this approach actually gives. Lets start from the general viewpoint of developers – the primary users of the framework.

The first simplicity achieved, is that there are only two conceptual elements developers have to deal with: the context and scenarios. Compared to the former approach, there were actually four (the context, scenarios, composite steps and individual steps), what made life of developers slightly more difficult. For each conceptual element any

developer usually has to know what are their purpose and limitations, so that he or she could be able to use them properly. It was also necessary to create many objects of different classes (scenarios and steps) in order to organize the code and handle it correctly, whereas in the current new implementation a developer only has to worry about scenarios.

Now scenarios are the primary means of abstraction and organization of the execution flow. It means that with the current new approach it is possible to build layered scenarios separating the overall logic into the abstraction levels, where the building blocks are always other scenarios. With the former approach this could also be possible, but would not be as elegant. It could be possible to build abstraction layers of individual steps, executing steps from within some other ones. However, the code in this case would not be readable and maintainable enough, because of the relatively low behavioral uniformity achieved. Low uniformity means that it could be technically hard to run scenarios, composite and individual steps in the same environment, which is because the were just designed to run in different environments. Scenarios of the former design approach had to be executed from within the presentation layer (at the user interface rendering time). Composite steps had to belong to scenarios and run within their environment, at the same time having individual steps to be executed at the lowest level. This means that the design statically defined three abstraction layers – scenario, composite step and individual step – where building additional abstraction layers would increase code complexity. Therefore, the benefit of the new approach is that there are no abstraction layers stated *a priori* by the design. All abstractions are achieved by manipulating essentially the only type of building blocks – scenarios and organizing code within them.

One more advantageous feature of the new design is the ability to link scenarios together. In other words, it is possible for scenarios to be aware of the previous “parent” scenario, which created them. Being aware of the parent scenario gives a convenient opportunity to make a backward operation. From the user point of view it means that on the screen there is a “Back”-button, which will lead to the previous screen. Thus, when user clicks the “Back”-button, the parent scenario is being invoked, which does all required context preparations, possibly runs some additional logic and renders the previous screen.

The “go-back” mechanism is common for all scenarios. This mechanism does not

depend on specifics of the implemented scenario, so it was organized into the abstract class as a utility function. Each individual scenario written by developer, and which extends this abstract class automatically gets this means for building “Back”-buttons. Thus, providing the “go-back” functionality for the user interfaces is even simpler process than making some other ordinary buttons with specific logic.

However, everything good comes at price. The next chapter is going to cover the trade-offs of this new conceptual approach to building user interfaces.

4.2.3Something we must sacrifice

The primary trade-offs of the new design are related to the system resources, such as operational memory and response time. Actually, these are the characteristics that could be better, but for some reasons they are not.

Lets take the first trade-off – the operational memory required. The systems built on top of the CuE platform user interface framework could consume less memory if there were no scenario chaining capabilities. Chaining is the ability to access parent scenarios discussed above. This new concept requires more operational memory, because it is needed to store the entire history of the scenarios a user walked through. This is required in order to provide unified functionality for implementing “Back”-buttons. The problem is that a user may decide to move back to the very beginning and to the very first screen. It means, that very often scenarios have to store some information in order to make steps back possible, which is why additional memory is required.

Of course, a user does not usually feel extra memory consumption so much, as the most important properties for him or her are system reliability and response time. There are no problems with reliability of the current new approach. However, some issues of system response time exist. Response time is the property the user may actually feel and with the higher response delays may experience some inconveniences.

System response time is critical, especially when many users are working with the system simultaneously. The obvious reason for that is the abstraction levels present in the user interface framework. Abstraction layers take their processing time for rendering screen or invoking lower abstraction layers. This extreme abstraction processing causes some delay that a user really feels. Luckily this delay is not so significant and is quite tolerable. However, it would be nice if the system response time characteristics could show better efficiency.

There is actually a third trade-off, which is still present since the former version of the user interface conceptual design. This trade-off is in dealing with large number of scenario classes. Yet unfortunately, no way was found for reducing the number of scenario classes. But anyway, the research is in progress and some day this issue will probably be solved as many others were.

To sum up, this new approach is considerably simpler than the former one. There are fewer types of objects developers should deal with, which makes its technical implementation a lot easier. The following class diagram (Figure 11) represents the new design.

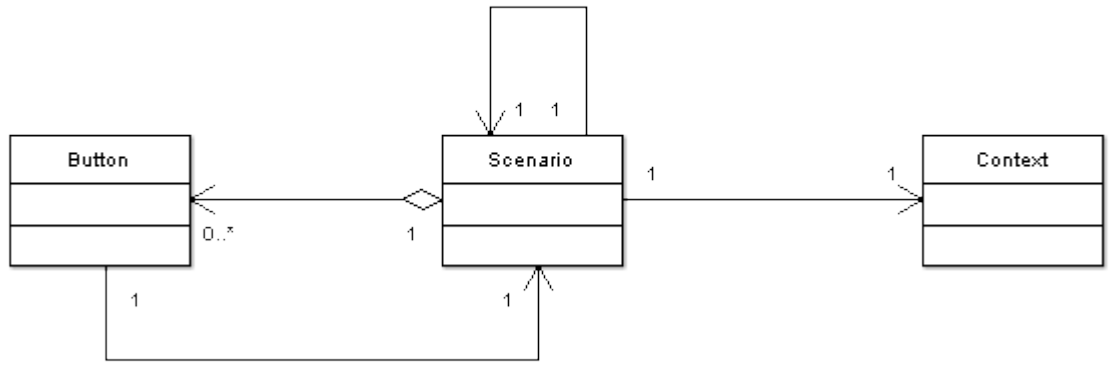

Figure 11: The third generation of the framework

Despite being that simple, the new concept has its own advantages and trade-offs. At the same time, they are not as critical as in the former implementation. But anyway, significant improvements over the previous version must be noticed and the new implementation may be considered as a better conceptual solution.

5The work to do

The implementation of the new conceptual approach cannot be considered the best one that is not needed to be augmented or improved in some way. With the changing world and business demands every piece of software eventually needs to be changed. But setting aside changing requirements, this new concept of the user interface framework still lacks something really important that would make a life of a developer many times easier.

5.1Describe navigational logic

The current concept for writing user interfaces allows easily distribute the navigational process into smaller executional elements – scenarios. Many of them usually make some preprocessing and end up with viewing some user interface screen. In general there is no problem with writing these separate scenarios and, by the way, this scenario programming became a lot easier. The basic problem here is at a little higher level – at the moment there is no way to get a higher level overview of the navigational logic.

Consider a situation, where a programmer is working on some specific scenario. The scenario actually may be linked with others in several ways:

- by means of buttons, which means that scenario is invoked when the button of the previous scenario is clicked
- by means of invoking the scenario directly as part of the logic of the other scenario
- by means of linking with the parent scenario and, therefore, having a back-link in the role of a “Back”-button.

It becomes very hard for the programmer to establish all relations of scenario he or she is working on with the rest of the scenarios. It requires additional activities, which essentially end up with looking into each scenario class and analyze the code. Thus, it would in no doubt made a dramatical negative impact into the development process, especially the construction phases.

An elegant solution might seem the following approach to building scenario-oriented user interfaces. The approach might consist of two stages: writing executable entities (scenarios in this case) and binding them together.

Writing executable entities is essentially achieved by the existing concept. However,

there is a little difference. With the current approach together with writing scenarios a developer writes bindings to other scenarios at the same time. It means that the two stages are messed up together in the current implementation. This might be useful to someday separate these two aspects.

After technically separating scenario writing and binding processes, a higher level of abstraction may emerge. This layer of abstraction would be responsible for the relations between scenarios such, that any developer could be able to see the entire picture of what is really going on with the navigation of the whole user interface. The navigational logic may be described in some kind of domain specific language, which could amplify the aspect of binding scenarios together. As a result, this would make the development of the user interface a lot more convenient and a fast process.

Therefore, describing navigational logic in one place is very important. Not only this would make the development easier, but also any possible further modifications would require less effort and become more reliable.

Conclusions

To sum up, the concept for the user interface development was introduced and its most important aspects were analyzed. Also, the whole process of the technological and conceptual evolution was concerned, revealing advantageous ideas and technical difficulties, which were met during the development of the framework itself as well as specific systems based on top of it. As a result, a convenient framework was born, which already found some use in the experimental project.

The present implementation provides great flexibility and reliability. Of course, to achieve these qualities some trade-offs had to be dealt with, but actually they were not so serious at all. The most important value achieved by this research in the area of the new conceptual approach to user interfaces was primarily the efficiency of the user interface development process, at the same time providing technical solutions and the design that would be convenient for developers.

However, the work is not over yet. There is still something to develop, what could dramatically improve the overall construction process of the user interfaces. This research gave a possible direction and a specific point of view on building user interfaces. This approach should further be analyzed and tested with many different kinds of systems and only its practical application may show how good it really is.

To sum up, the development of frameworks is often a very useful activity. Developing a technical framework for some conceptual idea is also a hard process that is time consuming and requires a lot of effort. But when it is done, when the technical approach obtains the natural look of the conceptual idea, the framework rises in the form very close to how humans really think and see the problem. Thus, having this in mind may in fact make software engineering significantly easier, what was successfully demonstrated by the current thesis.

References

[1] C. Bauer, G. King, *"Java Persistence with Hibernate"*, Manning, 2007, 841 pages.

[2] Elliotte Rusty Harold, W. Scott Means, *"XML in a nutshell"*, 3rd edition, O' Reilly, September 2004, 712 pages.

[3] Nicolas Kassem and the Enterprise Team, *"Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition"*, Sun Microsystems, October 3, 2000, 341 pages.

[4] R. Kent Dybvig, *"The Scheme Programming Language"*, Third Edition, The MIT Press, 2003, 295 pages.

[5] Peter Seibel, *"Practical Common Lisp"*, Apress, 2005, 528 pages.

[6] Joel Spolsky, *"The Law of Leaky Abstractions"*, November 11, 2002, URL=http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[7] Various authors *"JavaBeans"*, Sun Microsystems, August 8, 1997, 114 pages.

[8] Deepak Alur, John Crupi, Dan Malks, *"Core J2EE Patterns Best Practices and Design Strategies"*, Sun Microsystems, 419 pages.

[9] Brett McLaughlin, *"Java and XML"*, O'Reilly, 2001, 361 pages.

[10] Harold Abelson, Gerald Jay Sussman, Julie Sussman, *"Structure and Interpretation of Computer Programs"*, second edition, The MIT Press, 1996, 634 pages