Kroki: A Mockup-Based Tool for Participatory Development of Business Applications

* University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia
** Danulabs, Novi Sad, Serbia
mfili@uns.ac.rs, grist@uns.ac.rs, vladimir.marsenic@danulabs.com, darko.pejakovic@danulabs.com, igord@uns.ac.rs

Abstract—This paper presents Kroki (fr. croquis – sketch), a tool for participatory development of business applications based on mockups. Kroki provides a graphical editor for visual creation of mockups and two engines (web and desktop) for mockup execution. Kroki is developed in order to foster development agility, communication and better understanding of end-user needs. The mockup editor and engines are based on our EUIS (Enterprise User Interface Specification) DSL for specifying user interfaces of business applications.

I. INTRODUCTION

Mockup tools have become very popular for eliciting user requirements in participatory design. A mockup is a quick sketch that helps specifying many aspects of the user interface. Being in paper or electronic form, a mockup can be treated as a simple prototype that is useful in participatory design, for a better understanding of the users’ needs. Mockup tools are usually simple to use thus helping end users actively participate in the requirements solicitation process by using a simple and intuitive notation [5].

However, although mockups have a great potential, most approaches to software development use them only in the information gathering process, upon which the mockup is discarded. Besides spending time and effort on development artifacts that are thrown away, this can cause the difference between the end solution from the users’ original requirements [6].

This paper presents Kroki (fr. croquis – sketch), a tool for participatory development of business applications based on executable mockups. Contrary to the common practice, a mockup is used not only for requirements solicitation (and then discarded) but during design and implementation as well, being a basis for automatic execution and/or code generation of business applications. A mockup created with end users can be later supplied with implementation details through a set of views that support participation of developers with different specialties.

Since Kroki provides immediate execution of the application being sketched, it can significantly contribute to decreasing the communication gap between development team and users. According to the principles of agile development, information gathering is most effective if it is based on something that works [4].

Mockup creation and execution is based on our EUIS (Enterprise User Interface Specification) DSL (Domain Specific Language) for specifying the user interface of business applications at a high-level of abstraction [2]. Kroki actually implements this DSL’s concrete syntax that is designed to support modeling user interfaces (UI) in a “natural” way (visually resembles general-purpose mockup tools, see Figure 1). Mockup execution is provided by two aspect-oriented engines, for web and desktop applications, based on EUIS DSL (Figure 3).

Those two engines support all three tiers of a software application – the user interface, the business logic, and the database. Unlike most other solutions where only the UI skeleton is executable, Kroki’s executable mockups can be tested through all three application tiers.

Apart from the concrete syntax mentioned here, EUIS DSL has another two concrete syntaxes: the first is textual (also available in Kroki, enabling experts to quickly create mockups from the command console), and the second is graphical, implemented as a UML profile, intended to be used within a general-purpose UML modeling tool such as MagicDraw or Sybase PowerDesigner. Our goal for Kroki in its final form is to also provide graphical editor for EUIS notation based on UML profile, so that different specialists who take part in the development process can choose the modeling tool they prefer when end-user is not present. In its current form, Kroki provides read-only viewers of the application being sketched based on this notation. Viewers are also useful since they provide developers with persistence data model and user-interface model of the application, which are usually hard to comprehend just from UI sketches (see Figure 13).

The source code of Kroki, the examples presented in this paper and additional material can be obtained at [3]. Kroki is available under the terms of MIT license.

The rest of the paper is structured as follows. Section II reviews the related work. Section III presents the architecture of Kroki and briefly explains EUIS DSL. Section IV illustrates the use of Kroki. Section V concludes the paper and outlines future work.

II. RELATED WORK

While developing Kroki we have analyzed the papers and projects that are focused on mockup-based software development and requirements elicitation.

In [1, 8], mockups created by general purpose mockup tools are parsed, manually enriched with tags that add semantics to its elements and then translated to an abstract UI model that is further translated to presentation and navigation models of a web application. The content model can be derived as well, but it is not always precise enough since derivation is based on heuristics. However, after a set of transformation, developers can obtain an executable prototype from the mockup that user can try out, similar to our solution.
In [5] is presented UML language and its extension for specifying the user interface. UMLE consists of a family of languages based on simplified UML for specifying different aspects of an application. The most interesting part of UMLE for this discussion is the UI prototype generator named UIGU. The UIGU is based on many heuristics for transforming UMLE models to a usable UI. Main difference with approach presented in Kroki is that development process in [5] starts with the creation of “classical” models, while we start with UI modeling in terms of mockup creation.

Another approach to using mockups of user interfaces is presented in [6]. The tool presented in [6] takes mockups saved in files created by one of the supported general-purpose mockup tools and then performs its parsing and transformation into user interface program code for the chosen development platform. Mockups are used only to generate code that implements the user interface skeleton, while the rest of the application has to be implemented manually. The similar solution is also presented in [9].

The paper [7] presents the concept of window/event diagrams (WED) that represents a combination of user interface models and state charts. It is used to specify the layout and the behaviour of the implemented application. Each of the application's graphical elements (windows, dialogs, etc) that are sketched are represented as a state, while the events on those elements (mouse clicks, entering text into form fields, etc) initiate state transitions. This solution outlines one of the directions in future development of specifications dealing with the behaviour of user interface elements.

In [7], mockups are used to generate the code that helps user to investigate both the graphical layout of the application and the behaviour of the user interface elements. But it is still not a complete solution and the implementation of other aspects of the application (persistence layer, for example) requires manual intervention.

All presented solutions are using heuristics to produce some kind of model (UI model, presentation model, data model, etc) from the mockup or vice versa. This process is prone to errors and usually requires manual intervention to improve archived results.

III. THE KROKI ARCHITECTURE
A simplified idea for mockup-based development of business applications is presented in Figure 2.

We can generally assume that a sketched form can be mapped to a UML class, with its elements mapped to class members (each component to an attribute). With such mapping we can generate the skeleton of the UML class and of the user interface. This approach is used by most of the papers presented in the previous section.

The problem with this approach is that it is based on an automatic mapping of the mockup to the model, where the accuracy of mapping can be guaranteed only for the simplest of forms. For more complex forms usually present in business applications (such as parent-child forms, forms with calculated or aggregated fields, forms that need navigation to other forms, etc) this approach does not yield satisfying results. If we want a fully
functional running application that can be immediately tried out, then additional semantics is needed that general-purpose mockup tools lack.

Although Kroki visually resembles those general-purpose tools (see Figure 1), it can provide a fully functional application since it is based on EUIS DSL that has the needed semantics. Kroki implements the EUIS DSL concrete syntax in a way that looks like the mockup drawing to the user. Drawing is based on a number of types of forms and its elements whose layout and functionality are defined by our user interface guidelines. These guidelines, with some adaptations for different development platforms, is used since 1996. in more than a hundred applications, developed by several software companies that adopted our approach to business application development.

Our user interface guidelines are aimed to provide the simplicity of use of the application, quick user training, and automated development of the user interface. They are focused on specifying coarse-grained application building blocks (subsystems, forms, reports, etc) rather than fine-grained components, as practiced by most of the other existing standards (see [18] or [24], for example). Because of this, EUIS DSL enables developing models on a higher abstraction level, thus increasing development speed and improving the model readability. A brief overview of these guidelines is given in Section III.A.

Since the EUIS DSL metamodel provides a firm link between the persistence data model and the user interface model (see Sections III.B and III.C), mocking up will produce additional information needed to create the other tiers of the application (the database and the middle tier). Hence the sketched application is executable at any time, using a desktop-based or a web-based AOP (aspect-oriented programming) engine that directly implements EUIS DSL concepts (Figure 3).

Invoking a mockup will generate a middle-tier Java persistence layer using JPA (Java Persistence API) that is used to automatically create the database schema deployed in an embedded H2 database management system [16] that is included in Kroki. Other information contained in mockups are stored in configuration files that are used as inputs to AOP engines. Those information are used to dynamically create the remaining elements of the application (forms or pages, menus, etc). The end user can see and try out the application at any moment, enter or search for data, navigate through forms or pages, so that errors in specifications and unresolved issues are quickly detected.

Design of EUIS DSL, as well as the design of the user interface of Kroki, is aimed at facilitating the reuse of information gathered from end users during the specification phase and implementation phases in order to accelerate the development process and reduce discarding the artifacts. According to [12], the systematic reuse of artifacts across a development lifecycle can yield an order-of-magnitude improvement in development productivity.

The following is a brief overview (given here for the sake of completeness) of our user interface guidelines for business applications and the EUIS UML profile. Then, we will explain design of EUIS DSL metamodel used in Kroki.

A. Our User Interface Guidelines For Business Applications

We shall review here only those elements that are necessary to formulate the EUIS DSL. The more details can be found in [15].

A standard(ized) form is designed to present the data and all available operations on that data visibly on the screen so that the user can select a data item and invoke an operation on it. Standard operations (common to all entities) are available through the toolbar, while the entity-specific operations are presented as buttons at the right-hand side (see Figure 4).

Operations common to all entities supported by the standard form are the following: query by form, view, add, update, delete, copy, navigate, and change view (alternating between the grid view and a "row-per-screen" view). The specific operations include complex transactions that involve the entity, generating reports, and invoking linked (next) forms.

A standard panel has the appearance and the behaviour of the standard form but is embeddable into other complex forms. Standard panels are commonly used for building parent-child and many-to-many forms.

A panel can comprise fields that represent the attributes of the entity (editable by end user or read-only) or derived (aggregated, calculated, or lookup) fields.

Navigation among forms is performed using three mechanisms: zoom, next, and activate. A zoom is a mechanism for invoking the form that is associated with the entity connected with the current one by association, where the user can pick a value and transfer it back to the form where the zoom was invoked. A next is a mechanism for invoking “next of kin” forms. It provides the transition from a form associated with the parent entity to the form associated with the child entity in a way that the child form presents only the data filtered according to the selected parent (like parent-child but in separate windows.
or pages). An activate is a mechanism for invoking one form from the other without particular constraints. The activated form does not need to be of the same type as the activator form.

A parent-child form is used to display data with the hierarchical structure, where each element in the hierarchy is modeled as an entity in the database and is presented by a standard panel. A panel at the $n$-th level of the hierarchy filters its content based on the selected data item at level $n-1$.

A many-to-many form is used for quick editing of many-to-many relationships between the data items, with or without associated classes.

### B. EUIS UML Profile

Although used since 1996, in a number of our tools (for example, see [21, 22, 23]), the first formal implementation of EUIS DSL is carried out five years ago in the form of an UML profile [2]. The goal of this implementation was to provide a simple integration of user interface models with the models that represent other aspects of the application (the problem domain model, the persistence layer model, the physical database model, etc) by using UML as a common foundation. An integration of the user interface model with the rest of the application’s models is one of the challenges that prevent wider use of model-driven user interface development [19].

The EUIS profile contains a number of stereotypes based on presented user interface guidelines categories. Stereotypes are extensions of metaclasses Element, Class, Property, Operation, Parameter, Constraint, and Package from the UML::Kernel package. They can be grouped as follows (see Figure 5).

**Visible class (panel):** extensions of Class used for mapping of classes from domain model to different kinds of panels (as defined by the user interface guidelines: StandardPanel, ParentChild, ManyToMany, ParameterForm, etc). A standard panel is always associated with a persistent class that perform data-manipulation activities for it.

**Visible attributes:** extensions of Property that provide the mapping between the attributes of “visible classes” to UI components (Aggregated, Calculated, Editable, ReadOnly, Lookup, etc). Editable component is always associated with a persistent property (an attribute of a persistent class) for whom it provides data-entry and display operations.

**Validators:** an extension of Constraint metaclass used for modeling domain constraints. The next release of Kroki will feature automatic generation of constraints specified by OCL, based on Dresden OCL [20].

![Figure 4. An example of the standard form (table and “row-per-screen” view).](image)

![Figure 5. The EUIS UML profile.](image)
Visible parameter: an extension of Parameter that represents a parameter of a visible method whose value is entered in an UI component within a parameter panel associated with the visible method.

Element group: an extension of Property used for grouping elements of a visible class (attributes, methods, associations). A group is usually presented as a frame with a title or a tab in a tabbed panel.

Visible methods: extensions of Operation that denote methods with an associated button or that can invoke them. A method may be a general-purpose method or a method with semantics from the problem domain (Report, Transaction). Methods with the Transaction stereotype will be represented in the middle tier with a method invoked by the button. The method body has to be manually implemented.

Visible association ends: an extension of Property metaclass that can be applied to a navigable attribute belonging to a binary association between two “visible classes”. It defines the relationship between the panel associated with the class that has the attribute and the panel associated with the class that is at the other side of the association. The nature of the relationship is defined by the applied stereotype: Zoom, Next, Activate, Hierarchy (for parent-child panels), and Group (for arbitrary panel groups contained by another panel, without previously defined semantics).

Business subsystem: an extension of Package used for modeling business subsystems.

All aforementioned stereotypes inherit the VisibleElement, that is the extension of the Element metaclass. Since Element metaclass is a common superclass of all UML metaclasses, this facilitates the representation of all model elements with an UI component and a label, where applicable.

A simple example of developing a business application with the means of EUIS profile and our previously implemented tool in the form of a MagicDraw plugin [2] is presented in Figure 6 (a real world example is presented in [14]). Figure 6.a presents a part of a platform-independent model of the problem domain that models purchase orders (classes OrderHeader, OrderItem, Product i Customer). The problem domain model is automatically transformed to the user interface model (Figure 6.b) where each class is given the StandardPanel stereotype, each attribute – the Editable stereotype, each method – the Transaction stereotype, and each association end with 1 ili 0..1 cardinality – the Zoom stereotype.

Manual interventions on the UI model includes the class Order with the ParentChild stereotype that represents a parent-child form with OrderHeader being the parent and OrderItem being the child. Moreover, stereotypes of some attributes have been changed to Aggregated, Calculated and ReadOnly. Aggregated and Calculated attributes are supplied with the specification of how they are calculated.

After these manual interventions in the UI model we can generate code using the model-to-code transformation (Figure 7).

Similar steps are taken in order to create the middle tier – the problem domain model is automatically transformed to the persistent data model which, after manual intervention, is used to generate the middle tier code and the database schema (Figure 7).
C. EUIS DSL Implemented in Kroki

Contrary to our previous solution presented in Section III.B, Kroki is designed as a tool independent of general purpose UML tools with the main purpose of quicker and more effective inclusion of users in the development process. Although the previously described process yields executable prototypes relatively quickly, there is still a gap that has to be crossed when mentally mapping user requirements to UML models and then transforming them to UI models and to executable applications. Since the development team is working without the end users in that period, the probability of introducing errors and the potential number of development iterations needed to reach the final product is higher.

Developing prototypes using Kroki is performed directly using the terms that are familiar to the users (by drawing mockups) so the users can be included in the development from the start.

In order to keep the option of creating model views at an arbitrary level of abstraction, for supporting users and developers with different skills and specializations, we have kept the structure of the EUIS DSL metamodel based on a simplified UML::Kernel metamodel and its extensions for modeling UIs and persistence layers. Layout metamodel is added in order to provide design of mockup layouts (Figure 8). Layout metamodel structure is presented in Figure 9 (similar metamodel can be found in [6]).

This way, UI model elements (instances of “visible” metaclasses) can be viewed as mockups through mockup editor interface, but also as “ordinary” classes that can be viewed by UML viewers and, in next release of Kroki, designed by class diagram editors (Figure 10).

This way, the developer that models the business domain can only observe the class diagram, the end user can observe the visual specification of forms, the database designer can deal with the database schema and the mapping of the persistence layer (Figure 10), while the model is integrated at all times. Thanks to this no time is spent on transformations that usually slow development down and complicate the tool since the manual modifications to models have to be preserved [17].

Viewers for different kind of models are developed using PlantUML – open source tool for UML diagram drawing [25].

IV. USING KROKI

The Kroki user interface comprises the following parts (see Figure 1):

- Tree views for describing the hierarchical structure of business systems and subsystems and their parts (upper left corner).

- Property editor panel is divided into two tabs containing basic and advanced settings. The basic settings can be adjusted by the end user (labels, colors, layout mechanisms, etc), while the advanced settings are adjusted by the developer (data format, database mapping details, behaviour of user interface elements, etc). If the advanced parameters are not set, defaults are used, so that executing the application is always possible.
A sketched desktop or web application can be invoked by clicking the buttons on the main toolbar (Figure 12). If the application menu is not specified, the structure defined in the tree view component is used.

Right-clicking the package in the tree view will provide the display of the UI model, the persistence model, or the problem domain model. Right-clicking the project in the tree view will present the models for the whole application (Figure 13).

V. CONCLUSION

The research conducted in 2000 on a large number of software companies in America and Europe [12] claims that the majority of participants has identified the insufficient participation of users and badly specified user requirements or frequently changing requirements as the main reason for software project failures. Although the research is more than a decade old, these problems still exist. The problems in communication among the users and the development team frequently lead to the final product not looking or behaving as needed, even if all specifications presented by the development team are signed by the users [11].

Kroki is designed to help overcoming these problems by using mockups that can be executed and tested at any time. This is important since it should be possible to solicit and record information from the user based on something that works at any moment, according to the principles of agile development [4].

The user interface of Kroki and the structure of EUIS DSL as its foundation are designed to keep the information solicited during system specification in later stages of design and implementation. This facilitates the development process and reduces the information loss. Although still at a prototype stage, our first experiences show that we are on the way to achieve those goals.

Compared to our previous solution [2] the advantage of this proposition is that the application model is integrated the whole time, thus avoiding transformations that usually slow development down and complicate the tool itself [17]. Additional benefit is that modeling the UI relies on a notation understood by users thus improving the communication, and that modeling form layouts is included as well contrary to the previous solution that relied to manual code alterations after generating the UI.

We plan to develop Kroki further in order to (1) integrate Kroki with a class diagram editor for supporting developers that prefer this way of working; (2) create a language for sketching business processes with end users and integrate it with the EUIS DSL and Kroki; (3) provide interoperability with general purpose UML tools.
by importing and exporting models; and (4) integrate Kroki with Dresden OCL library for automatic generation of OCL constraints.

Besides, we plan to conduct a survey to measure the cognitive load of users, the development speed and the percentage of errors made in the requirements solicitation phase when using Kroki and compare them to the use of “classical” specification documents and techniques.

ACKNOWLEDGMENT

This work is partly funded by the Grant No. III–44010 of the Ministry of Education and Science of the Republic of Serbia.

REFERENCES